ARMY RESEARCH LABORATORY

Design, Development, and Demonstration of a Prognostics and Diagnostics Health Monitoring System for the CROWS Platform

by Marvin A. Conn, Gregory Mitchell, Derwin Washington, Andrew Bayba, and Kwok F Tom

ARL-TR-5206 **June 2010**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-5206 **June 2010**

Design, Development, and Demonstration of a Prognostics and Diagnostics Health Monitoring System for the CROWS Platform

Marvin A. Conn, Gregory Mitchell, Derwin Washington, Andrew Bayba, and Kwok F Tom
Sensors and Electron Devices Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2010	Interim	FY08–FY09

4. TITLE AND SUBTITLE
Design, Development, and Demonstration of a Prognostics and Diagnostics Health Monitoring System for the CROWS Platform

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Marvin A. Conn, Gregory Mitchell, Derwin Washington, Andrew Bayba, and Kwok F Tom

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-SER-M
2800 Powder Mill Road
Adelphi, MD 20783-1197

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-5206

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Automated data acquisition has become a major part of the military's prognostics and diagnostics program as it moves towards a condition-based maintenance approach. The desire to apply this to a majority of new and legacy systems has led to the development of a prototype Prognostics and Diagnostics Health Monitoring System (PDHMS) with both serial and wireless communications capabilities that can be easily configured to different mechanical and electrical systems. This report addresses a complete system architecture of a prototype PDHMS. It verifies the data collection capabilities of the PDHMS by obtaining vibration signatures from bearings running on a machinery fault simulator, and by integrating the PDHMS for an embedded system-level demonstration in the Common Remotely Operated Weapons Station (CROWS). This report addresses the challenges in designing the hardware and developing the firmware for the PDHMS, discusses system limitations, and suggests areas for future improvement as seen by the developers.

15. SUBJECT TERMS
Prognostics diagnostics microcontroller wireless I2C data acquisition RF tag

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Marvin Conn
Unclassified	Unclassified	Unclassified	UU	88	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-0823

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

We would like to thank digital designer Mr. Russ Harris for performing the board layout and assembly, and for giving invaluable feedback on design approaches and options.

1. Introduction

As the U.S. Army continues towards a condition-based maintenance (CBM) approach for logistics and mission readiness, the need for automated data acquisition becomes paramount for success. The analysis of critical system data minimizes the vulnerabilities of combatant forces, maximizes the availability of combat ready equipment, and concurrently produces a proactive logistics enterprise. This report discusses the performance of a Prognostics and Diagnostics (P&D) Health Monitoring System (PDHMS) designed for remote data acquisition in a variety of Army systems. The PDHMS, developed at the U.S. Army Research Laboratory (ARL), uses an onboard microprocessor, transceiver, and a variety of sensors to monitor key points of interest within a platform and transfer data while remaining transparent to the end user.

The hardware is based on a highly configurable design with the capability to monitor electrical and mechanical systems. We plan to demonstrate the flexibility of the PDHMS architecture on both an electrical system and mechanical system: electrical fuses within the Combat Remotely Operated Weapons System (CROWS) and mechanical bearings for use in ground vehicles. This report compares experimental vibration data for mechanical bearing degradation collected by the PDHMS to data collected by an off-the-shelf data acquisition system.

In documenting the capabilities of the PDHMS, we cover the hardware and software architecture, as well as the graphical user interface (GUI) developed to configure the PDHMS to remotely issue commands to all devices within the PDHMS network and display the results graphically. This report also covers any observed shortcomings of the present design and makes recommendations on what future implementations of this design might look like.

2. System Design Concept

The PDHMS design concept focuses on having one or more microcontroller-based Prognostics and Diagnostics Sensor Modules (PDSM) or PC boards designed to take measurements on key system test points in the CROWS. PDSMs acquire and store sensor data to their local memory. Each PDSM can communicate between PDSM nodes as well as communicate back to a central Prognostics and Diagnostics Control Station (PDCS). The PDCS remotely configures and queries the PDSMs. The combination of multiple PDSMs and a single PDCS makes up the PDHMS. Figure 1 shows the overall system design concept. Connected to each PDSM are the required sensors to monitor test points of interest. To support such flexibility, the PDSMs must support multiple mediums of communications such as wireless, wired, and universal serial bus (USB) connections, which provide users reasonable flexibility. The general operating concept of

this design is that the operator located at the PDCS establishes a remote connection to each PDSM through either wireless or serial wire mediums. The user at the PDCS then issues configuration commands to each PDSM. Once the operator has configured and activated the PDSMs, the PDSMs operate autonomously.

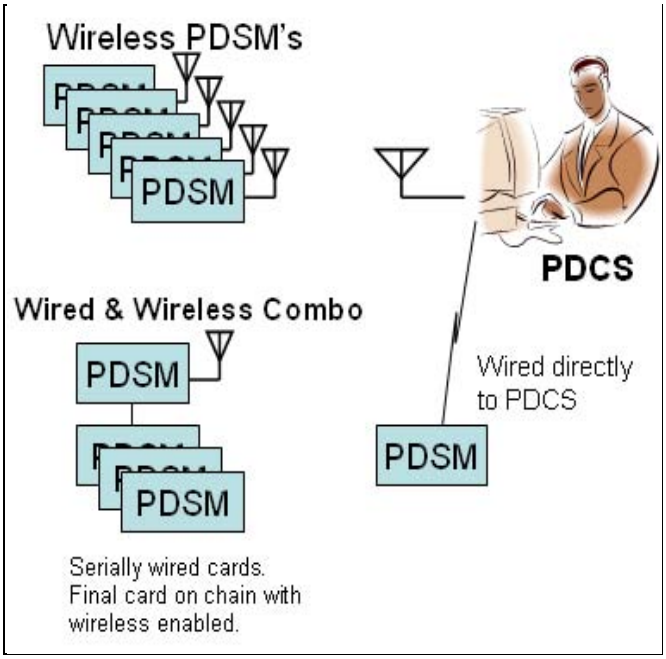


Figure 1. System design architecture.

Once the general design architecture was complete, we looked at further defining the sensors required for the PDSM to prepare it for a demonstration of the system installed into the CROWS platform, as shown in figure 2. The demonstration would encompass monitoring four separate circuit cards, controlling the azimuth, elevation, sensor unit (SU), and linear actuator, located in separate cavities of the CROWS. In each cavity, the requirements were to monitor temperature on a Polymer Positive Temperature Coefficient (PPTC) Resettable Fuse; temperature on a Pulse Modulator integrated circuit (IC); the main power supply voltage and current; and the three-axis vibration characteristics of the four system drives. These requirements resulted in the final PDSM design comprising the following sensor capability: three thermocouples sensors, one voltage sensor, one current sensor, one external three-axis accelerometer, and one onboard accelerometer.

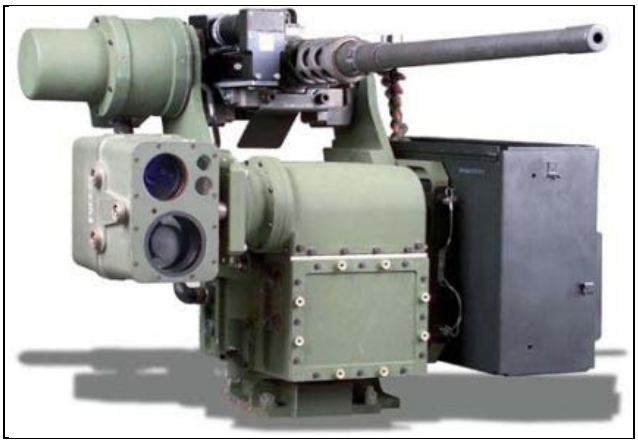


Figure 2. CROWS.

3. PDSM Board Design

The core of the PDHMS design effort focused on the design of the individual PDSM boards. This section gives a more detailed description of the design process for the PDSM used in the final CROWS platform demonstration. A photograph of both sides of the PDSM is shown in figure 3. The dimensions are 4 in by 2.125 in. These dimensions were driven by the requirements to install the PDSM into the CROWS; there is a capacity to shrink future designs, if necessary. Also, because the type of application drives the number and type of sensors in the PDSM design, the size limitations of the design are application specific in some respects. In future redesigns, tradeoffs may have to be made between performance, types of sensors allowed, and overall PDSM size.

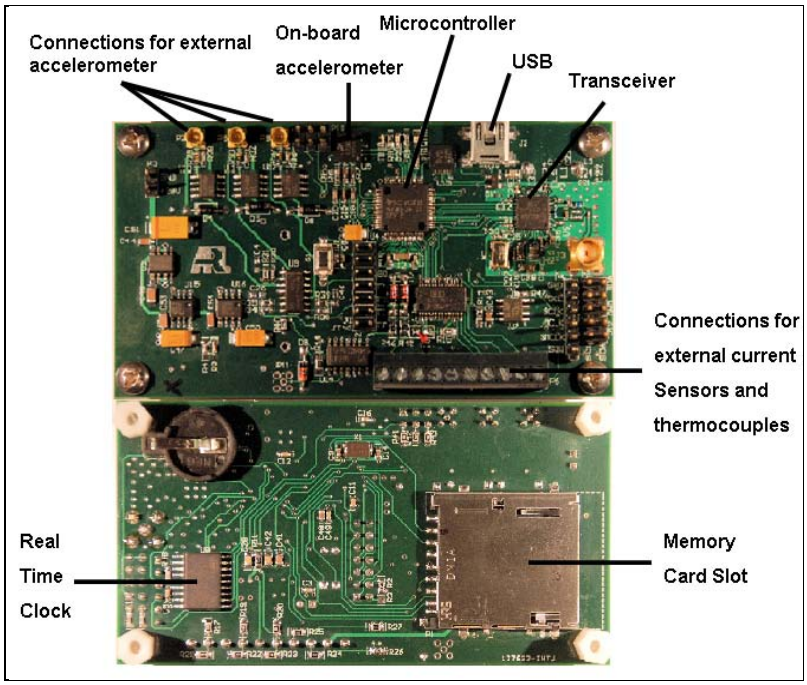


Figure 3. Top and bottom layout of the PDSM highlighting key design elements.

The Texas Instrument (TI) MSP40F2619 microcontroller was used in the design. This MSP430 has 128 Kb flash and 4 Kb random access memory (RAM). The MSP40F2619 memory was adequate for this demonstration, but the small RAM size limited the number of continuous samples that could be acquired during acquisitions. In this application, the RAM space had a general allocation of ~1024 bytes for sensor sampling and the remaining 3072 bytes for general firmware logic, which limited the contiguous blocks of samples to 2 bytes per sample, resulting in 512 samples per acquisition block. The small RAM size is a problem for applications that require larger data acquisition blocks. A more efficient transfer of the data to a secure digital (SD) memory card would help mitigate the limited RAM size, but a larger RAM capability is highly desired.

The PDSM is powered by a 30-V power connector. Although the PDSM board is low power and the MSP430 microcontroller unit (MCU) can run off of 3.3 V DC, the 30-V power connector was designed to allow the PDSM to accept 30 V supplied from the CROWS. Also, the external three-axis accelerometer requires a 24-V power source, which is derived from this 30-V input. Onboard the PDSM, the 30 V is regulated down to 24 and 3.3 V and distributed to the circuit components. There are three miniature coax-M connectors to connect the Model 3000 (M3000) external accelerometer. The connectors are for the *x*-, *y*-, and *z*-axis of the M3000. Each connector was fed to the required conditioning circuitry for the M3000 and the analog-to-digital converter (ADC) inputs of the MSP430. We noticed in the lab measurements that when the M3000 was not physically connected to the PDSM through the miniature coax connectors, the

voltage levels feeding the MSP430 ADC12 were driven above the MSP430's rating of 3.3 V. This problem must be fixed in the next design.

3.1 Communications Mediums

The three ways in which the PDSM boards communicate are wireless, inter-integrated circuit (I2C), and USB. A user can issue commands to the board to configure the board or retrieve the board status or measurements data from any one of these communications mediums. The manner in which they are used or configured is strictly a matter of how the firmware is written. The TI CC2420 2.4-GHz RF chip provided the wireless communication capability to the PDSM boards. An I2C bus connection was available to link multiple boards together for communication of data between one another. The USB provided an ability to connect the PDSM board directly into a laptop or desktop computer.

Since the boards were designed to operate in a networked configuration, a method was required to identify each board uniquely. We accomplished this by using a three-port jumper to set the PDSM local node address. The jumpers allowed us to set addresses from 0 through 7, providing a maximum of eight possible PDSM nodes in the demonstration network. Theoretically, 65536 of nodes could be supported by either increasing the number of jumpers to 16 or by using some other means of control in the firmware.

3.2 Sensors

A screw terminal was used to connect the current, voltage, and three K-type temperature sensors. This terminal can handle a maximum voltage of 43.75 V, which should not be exceeded. The current sensor input was designed to use the CSA-V1 Hall Effect current sensor device. The maximum input on the current sensor input should be no greater than 2.5 V. A key problem with the present design was that there was no protection circuitry on any of the sensor inputs. During use, we damaged several PDSM boards as a result of misconnecting the voltage input on the power supply and voltage sensor input. To make the design more robust to inputs that may exceed design limits, protection circuitry must be addressed. Additionally, the screw terminal is not an ideal way to connect and remove the sensors from the P&D board. Investigating a better way to do this should be addressed in the redesign.

The card contains a MMA7260Q three-axis accelerometer. This accelerometer is used to measure the vibrations of the platform to which the PDSM is mounted or the orientation of the card (and the equipment in which it is installed) as other measurements are being taken to correlate measurement behavior with equipment orientation. An external trigger input was provided to allow the samples of the sensors to be synchronized with an external rising edge trigger input. The input on the line should read 0 to 2.5 V. Firmware for this feature was not implemented.

The PDSM shown in figure 3 illustrates the locations of the real-time clock (RTC) as well as the SD memory card. The ST M41T93 serial peripheral interface (SPI) bus RTC chip is used to time stamp the acquired sensor data for post analysis. A coin cell battery, such as the CR1220 3 V battery, can power the RTC when the 30-V power is not available. Upon removing power from the PDSM, the clock loses the time, so this problem must be resolved. To store the sensor data as it was acquired, a SPI SD/multimedia card (MMC) memory card was used.

Figure 4 shows the PDSM board with all sensors attached: three thermocouples, one external accelerometer, one current sensor, and one voltage sensor. The thermocouples are adhesive stick-on type so that they adhere to the surface of interest. The accelerometer is attached using miniature coaxial cables for each of the three axes. The current sensor is attached through a twisted pair of red, blue, and green wires. The voltage sensor is simply a thin gauge of wire that can be attached to the point of interest.

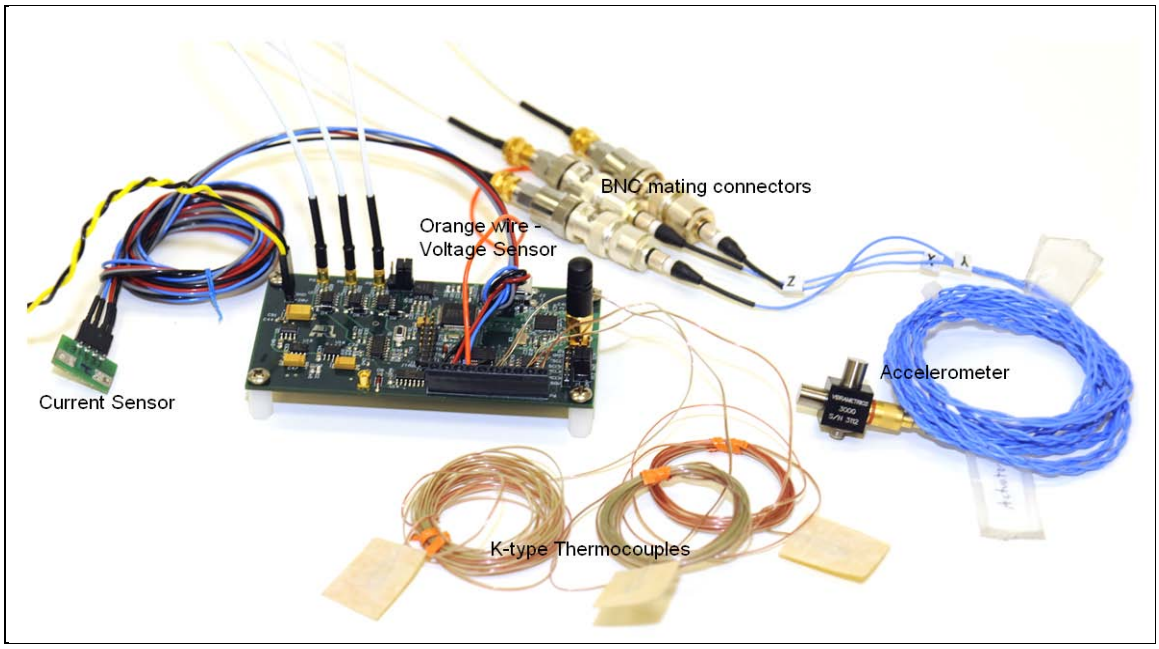


Figure 4. PDSM card with all sensors connected.

When making the connections, several problems become apparent and are viewable in the photo. First, there are many wires coming off the board to connect the external sensors. This may become a problem if the area in which the PDSM board is installed is very tight. Secondly, the screw terminal connector is not easy to work with, especially when frequently connecting and disconnecting sensors. Some form of a quick release terminal connector should be investigated. Also, there is a possibility that a sensor may inadvertently disconnect while in use, possibly due to system vibrations. A method must be put in place to automatically detect and inform the user when a sensor is no longer connected to the PDSM board. This type of functionality will likely involve some combination of hardware and firmware implementation.

Table 1 shows the major components used on the PDSM board design. All of the components used are commercial-of-the-shelf (COTS) devices. One key point to note in the parts list is that since this design, TI has developed MSP430 families that have integrated the wireless and RTC components on the MSP430 chip. These external components can likely be removed saving real estate and power in future PDSM designs.

Table 1. Major PDSM hardware components.

Part Description	Manufacturer	Part Number	Design Limits
Ultralow-power mixed signal microcontroller	Texas Instruments	MSP430F2619	16 MHZ 120 KB + 256 B flash memory, 4 KB RAM
External low noise three-axis accelerometer	Vibra-Metrics	M3000	±500g's
Onboard three-axis accelerometer	Freescale	MMA7260	±90° per axis
SD/MMC 2 GB flash memory card	Sandia	2 GB Sandia SD Card	2 GB Memory
RF 2.4 GHz IEEE 802.15.4 transceiver	Texas Instruments	CC2420	250 kbps ~60 mw
Real -time clock SPI	STMicroelectronics	M41T93	time stamp data
Hall Effect current sensor	GMW Associates	CSA-1V	High power current measurement ns ±45 A
External K-Type thermal couples (32–2282 °F)	General Electric	RL0503-5820-97-MS	32 °F (0 °C) to 300 °F (150 °C)
MSP430 ADC12 – 8 channel, 250 ksps multiplexed ADC	Texas Instruments	Part of MSP4302619	~100 ksps at 512 sample blocks
24 bit ADC, 15 Hz, 8 channel, differential	Burr Brown/Texas Instruments	ADS1241	15 sps, used for thermal couple measurements

4. Firmware Documentation

The firmware for the PDSM design was developed in embedded C/C++ using the MSP430 IAR Embedded Workbench software development environment, release 4.2.01. No operating system was used on the MSP430 microcontroller. However, because of the complexity of the required communications and the multiple tasks that the processor has to perform, a real-time operating system (RTOS) should be considered for future implementations of the PDSM design. The software for the PDHMS GUI was developed using the Microsoft Visual Studio Development Environment, 2005, version 7.0.9955, and used the MATLAB R2008b display engine. Various portions of the software and firmware were obtained from various sources of publicly released software as indicated in the source code. A CD is included with this report, containing all software and firmware required to implement the functionality as described in this report (see the appendix for details).

The descriptions for all the MSP430 firmware are located in the **IAR Embedded Workbench\pd-develop** subdirectory on the accompanying CD. This section provides a directory listing of the firmware followed by a brief explanation of its content and a statement of any recommended future development.

Pd-develop

Setting files are contained in this directory, which holds project settings defined by the IAR workbench compiler. These file names, "pd0develop.*" were all generated by the IAR workbench compiler and should not be modified without using the IAR development tool.

Main.c

This directory defines the top level firmware execution entry point for the functionality of the PDSM. Presently, this firmware contains too much functionality, which must be restructured such that the operations are defined by the programmed library firmware.

Adc12-lib

This directory contains the driver code for controlling the MSP430 ADC12 ADCs. These devices are used for acquiring data from the following sensors: the M3000 three-axis accelerometer, the current test point, and the voltage test point.

Adc1240-lib

This directory contains drivers for the eight-channel ADC1240 chip. This chip is used to make the three thermocouple measurements. Each thermocouple requires the use of two channels. One channel is used as a ground reference during measurements and the other channel is used for taking measurements on the thermister, which is used as the cold-junction temperature reference.

Cc2420-lib

This directory contains the device driver for the wireless CC2420 chip used in the design. This driver is used to support wireless communications. Although CC2420 can support the ZigBee protocol, this driver does not implement ZigBee. Future development could include the use of the TI ZigBee stack to make the wireless communications more robust.

Clock-lib

This directory contains a library to control the MSP430 clock frequencies. We attempted to implement a common library for setting all clock frequencies of the MSP430. This library controls SCLK, MCLK, etc. All device drivers in this development should use this library when required to alter clock settings.

Debug

This directory contains the IAR compiled code and debug information. Depending on the compiler settings, it will contain informative text files on memory usage of the complied firmware.

Docs

This directory is a repository for all documentation related to the design project, including this document.

Dosfs-1.03

This directory contains the FAT32 driver - fat32 library for performing input/output (I/O) on the SD memory card. This library is not used in the project, but needs to be replaced by a more robust and complete commercial driver, which is very likely to be part of a selected RTOS.

Efl-dosfs

This directory contains the FAT32 driver, which is just another possible free alternative for FAT32 file I/O and is here for documentation purposes only. It was not integrated into the design nor has it been tested.

Errorlib

This directory contains the library for generating error messages when system errors occur. Our aim was to develop a common library for all possible errors that can occur in the system for the purpose of communicating error status information back to the user either by light emitting diode (LED) flash patterns and or by sending messages back to the GUI interface. Note: Error messages are also contained in the cmdmsg.h file.

Fat32

This directory contains the FAT32 driver, which has not been tested or used. This driver is just another possible free alternative for FAT32 file I/O and is here for documentation purposes only.

Fileiolib

This directory is a higher level application programmer interface (API) that uses the dosfs-1-03 library to control read and write accesses to the SD memory card. Our aim was to make it easier to use the dosfs-1-03 by hiding low-level call details.

Globals

This directory contains the global.h include file, which contains global flag variables used primarily for passing status information regarding the communication interfaces (wireless, I2C, universal asynchronous receiver-transmitter [UART]). This approach needs to be redesigned. A better approach would be to eliminate the use of global flags and communicate this information in another manner, such as a COTS RTOS, which would be a lot easier to use.

I2clib

This directory contains the I3C driver library for I2C communications. This library implements the standard I2C communications protocol, allowing all nodes in the system to operate as either master or slave, and switch back and forth between the two modes as appropriate. This library needs to be enhanced to make it more robust for dealing with the I2C bus collisions that can occur when two or more nodes attempt to access the bus at the same time.

Intlib

This directory contains the interrupt library. This library places all system interrupt routines in one location so that the developer knows where to look for interrupt routines for development and debugging. There may be cases where some interrupt routines are not located here, but an effort has been made to place them into this library. This library is likely to change dramatically with the use of an RTOS.

Ledlib

This directory contains drivers to control access to the system LEDs. The LEDs provide minimal communication to relay status of the PDSM by blinking and lighting specific LEDs on the board. This library defines led color definitions, and blink count definitions

Lpm-sleep-lib

This directory contains some basic functions for placing the MSP430 into low power sleep for specified time periods to conserve power. This function is not yet used extensively throughout the design because in some cases during the development when the MSP430 was placed into sleep mode, it caused the MSP430 to hang. Placing the design into sleep mode for purpose of conserving power needs to be investigated extensively for future development on the design. Using an RTOS should make this process easier.

M41t93-lib

This library is used for setting and reading the M41T03 RTC chip. In the present release, the clock functions operate well; however, for an unknown reason, the clock value becomes corrupted if the main power to the PDSM is removed because the clock loses its time. This problem needs to be fixed in the next revision, or an MSP430 with an internal clock could be

used to replace this one, which is likely a better alternative. The problem is very likely due to a hardware design problem, and in particular, how the battery is powering the chip.

Math-lib

This directory contains the math-related routines. Note: There is a non-ported Fast Fourier Transform (FFT) library in this directory that may not be appropriate for the MSP430. All math functions on the project should be moved to this directory. For example, there are many math functions in main.c file that should eventually be placed here.

Matlab-tools

This directory contains the MATLAB simulation tools for the temperature and current sensors, including the routines that generated the C/C++ tables used in the therm-cup-lib routines. These tools are critical for interpreting the current and temperature sensor measurements. This directory also contains some miscellaneous example MATLAB routines that were, at one point, used to read raw data files generated from the ADC1240 library.

Mma7260q-lib

This directory contains driver code for the onboard MMA7260Q three-axis accelerometer, including the code to configure and read the MMA7260Q. In working with the MMA7260Q, it appears that occasionally the readings were totally erroneous, for reasons yet to be determined. A possible reason could be that the other activities on the MSP430 messed up the MMA7260Q settings. This problem requires further investigation to fix.

Msglib

This library implements the top level communications API for the design and is probably the most critical for long-term development. This library makes use of the lower level hardware access layer communications device drivers for I2C, wireless CC2420, and UART. This API defines the highest layer for implementing the message architecture for inter-PDSM message communications. This library performs fairly well when communication is taking place between only two nodes. Communications errors tend to occur when three or more nodes communicate at the same time. Some limited error corrections have been implemented, but generally the underlying drivers need to be made more robust for multimode node communications. The low level communications device drivers and a commercial RTOS should be used to make this overall communications system more robust. Communications collision detection and avoidance algorithms should be considered as well. ZigBee would address this concern for the wireless portion.

Node-address-lib

This is a simple library used to determine the PDSM's node address based on the PDSM's jumper settings. The node address is used by the msglib for communications.

Release

This directory is the location of the IAR compiled release code; no debug information is provided.

Sd-mmc-lib

This directory contains the low level SPI-based SD memory card device driver. Future implementations need to separate the SPI code out from this driver, and create a **spi-lib** directory to exclusively contain code for talking to the SPI interface for all SPI devices. Future implementation of the driver also needs to support direct memory access (DMA) storage for processor parallel operations.

Therm-cup-lib

This library contains device driver functions for taking readings on the attached thermocouples and provides routines for converting the readings to scaled temperature readings. Much of the code in this library was derived from the simulation code contained in the **matlab-tools** directory.

Usartlib

This directory defines low level hardware access layer code for universal synchronous/asynchronous receiver/transmitter (USART), which is presently used for the USB/USART connection so the GUI can pass messages into the system. A limitation with this library is that the present hardware design does not implement some sort of hardware handshaking control lines in USART communications. Although it may not be needed, implementation of hardware handshaking control lines may need to be considered to guarantee more robust communications on the USB/USART interface. For example, because there is no hardware handshaking, the GUI can potentially push more data across the USB than the PDSM can process. To address this, the GUI code has been designed to "pace" how much data it pushes to the board by delaying its writes to the USB. For robustness, this area should be addressed.

Utils

This directory contains the general utility functions, which are functions that may be required by other libraries or to do some sort of general processing tasks.

MSP430 241x,261x examples

This directory contains useful TI code examples for programming MSP430 peripherals. These examples are not integrated into the PDSM design. They are left here as a reference for future development.

5. Software Documentation

The software of the all of the GUI development is located in the **Visual Studio\progdiag** subdirectory. The following is a listing of the software directories with a very brief explanation of their content and a statement of required future development.

Cmdctlgui

This directory contains the PDHMS GUI source code. The GUI was developed using Visual Studio 2005 C/C++. Further development will use a later version of Visual Studio and likely be converted over to C# to take advantage of its GUI development features. When recompiling this code, it is important to make certain that the directory path location of the MATLAB plot libraries discussed below is properly coded into the source. This is a portability issue in the present software version that needs to be resolved. A key problem with this GUI interface is that it is not scalable if many (hundreds of) nodes are added into the system. The GUI should be redesigned with this in mind.

Matlab

This directory contains the MATLAB display routines used by the GUI code. Do not alter any of these routines without first understanding the impact on the PDHMS GUI code contained in the **cmdctlgui** directory. MATLAB was used primarily for rapid prototyping of the displays. For future development, it is desirable to eliminate the use of MATLAB for the data displays and to focus more on how data and processing results should be presented to the end user in an actual system.

Raw-device-reader

This directory contains a GUI for reading and displaying the contents of data on memory storage devices such as an SD memory card. This GUI has been used primarily for debugging during development on the memory the SD card and is used as a tool for development.

6. Data Acquisition Design Decision

A round robin technique was used in the data acquisition system for simplicity of implementation. For example, if during an acquisition, we wanted to sample from the external accelerometer the *x*-, *y*-, and *z*-axes and also from the voltage sensor, a block of samples from each input would be sampled and then stored to memory. This cycle would continue until a stop command was issued. In the present release of the firmware, a maximum of 512 samples could

be acquired. The reason for the simplicity of this implementation becomes apparent when considering the following discussion.

This discussion is meant to illustrate the complexities that would need to be addressed in the future implementation of a more sophisticated data acquisition scheme. A more ambitious requirement could be to simultaneously sample all sensors while simultaneously storing the data to the SD memory card without a time break in the data block sizes. The storage rate to the memory card would have to support the sum of the maximum sampling rates of all sensors. This would require use of the MSP430 DMA and likely require a typical scheme of ping ponging between two memory buffers while acquiring and storing. Key design considerations would be the MSP430's clock rate, the collective maximum sampling rates, I/O contention, RAM, SD card, and I/O speeds. Since the MSP430 controls all of these functions, one would need a clear understanding of what the system's acquisition requirements so that they can fit within the capabilities of the MSP430.

In extending this complexity to the present hardware, the following assumptions can be made with respect to possible sensor sampling requirements. For the three thermocouples, 2-byte words per sample at very low data rates of 1 Hz or less would be needed. The external three-axis accelerometer requires 2-byte sample words on each axis with a maximum sample rate of about 8 KHz per axis. The onboard three-axis accelerometer with max output data rate of 400 Hz each axis requires 2 bytes per sample. The current and voltage sensors will be assumed to sample at 8 KHz rate at 2 bytes per sample. Table 2 summarizes this discussion.

Table 2. Overview of different sensors used in the PDSM.

Sensor Type	Bytes Per Sample	Required Sample Rate (Hz)	Data Rate KB/s	Measurement Device
M3000 axis-x	2	8000	16	ADCMSP430
M3000 axis-y	2	8000	16	ADCMSP430
M3000 axis-z	2	8000	16	ADCMSP430
CSA-V1	2	8000	16	ADCMSP430
Voltage TP	2	8000	16	ADCMSP430
LIS302DL axis-x	2	400 (8000)	0.8 (16)	ADCMSP430
LIS302DL axis-y	2	400 (8000)	0.8 (16)	ADCMSP430
LIS302DL axis-z	2	400 (8000)	0.8 (16)	ADCMSP430
K-Thermocouple 1	2	1 (0.1)	0.02	ADS1240
K-Thermocouple 2	2	1 (0.1)	0.02	ADS1240
K-Thermocouple 3	2	1 (0.1)	0.02	ADS1240
		Required Storage Data Rate	82.5 (128)	

Several points can be made regarding the different sensors used in the PDSM. First, the MSP430 would have to time share its ADC12 ADC converter across the external accelerometer, the current sensor, the voltage sensor, and the onboard accelerometer. The MSP430 would have to manage switching across these sensors while maintaining the desired sampling rates across each sensor. As noted in table 2, all sensors do not have the same sampling rate, and conceivably the

user might have an interest in using sampling rates different from those in table 2. The MSP430 would have to initiate samples taken on the thermocouple sensors, and these sensors are sampled using the ADS1240 ADCs, which are SPI controlled. The MSP430 would have to direct the acquired data into the memory card on the SPI bus. The complexity of such an implementation soon becomes apparent, and one has to consider that such a configuration may not be possible with the MSP430.

7. Communications Hardware Design Details

7.1 I2C Design Details

The I2C protocol is a wired serial communications interface standard. Data are transferred on the serial data line (SDA) and synchronization is maintained by the serial clock (SCL). Each PDSM board can act as either an I2C slave or an I2C master on the I2C bus as implemented with the PDSM boards.

In figure 5, the I2C bus header P8 is used to interconnect two or more boards on the I2C bus. To make the connection, the SDA, SCL, and ground pins of each board must be interconnected using the P8 connector. On each board, all SDAs must be connected together, all SCLs must be connected together, and all grounds must be connected together.

Further, as shown in figure 5, the master node, the node with ID jumpers set to 0, has the two pull-up resistors, R13 and R12 jumpers, installed to pull the SDA and SCL lines high. On the master node with ID set to 0, a jumper is installed connecting pins 1 and 2 on P12, and a jumper is installed connecting pins 3 and 4 on P12. All other boards, PDSM nodes with jumper ID set to 1 through 8, do not have the P12 jumpers installed. Figure 6 shows that MSP430 pins P3.1 and P3.2 are used to control the SDA and SCL signals, respectively.

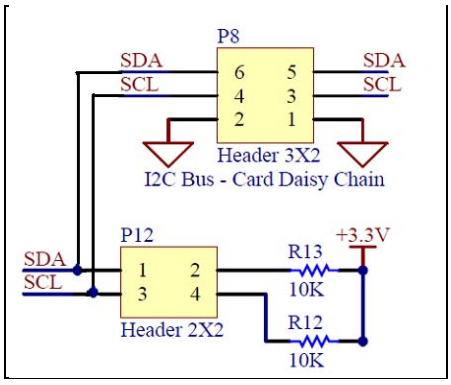


Figure 5. I2C schematic.

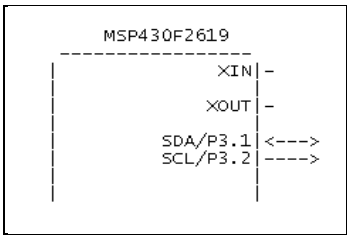


Figure 6. I2C interface to MSP430.

7.2 USB Design Details

Figure 7 shows the schematic of the USB interface design, which uses the CP2102 USB to UART bridge chip. The present implementation does not implement any hardware handshaking, which may be of interest in future designs. Figure 8 shows the interface connections of the URXD0 and UTXD0 control lines to the MS430. MSP430 pins 3.4 and 3.5 connect to the CP2102 pins 25 and 26. The USB interface provides the communications interface between the PDSM board and a laptop or computer workstation. On a Windows 2000 or XP platform, the device driver CP210x_VCP_Win2K_XP_S2K3 from Silicon Labs must be installed on the laptop or workstation that runs the GUI. A USB connector connects at J2 for direct PC to PDSM communications.

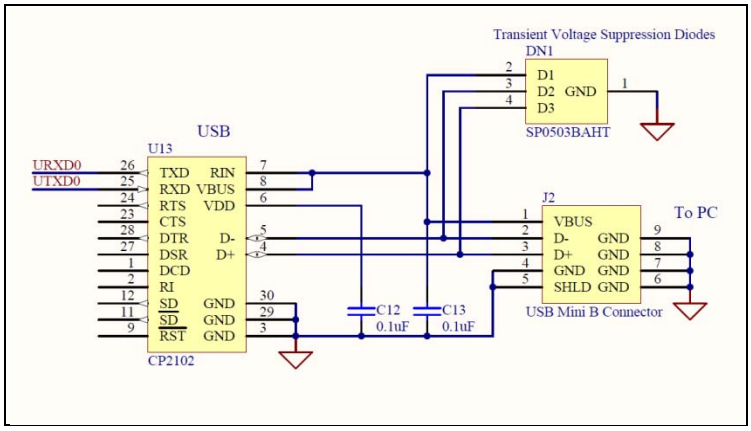


Figure 7. USB schematic.

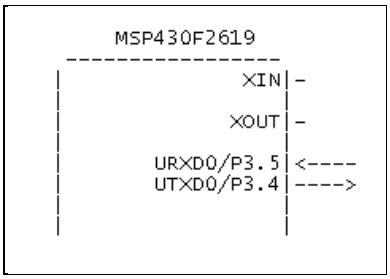


Figure 8. USB interface to MSP430.

7.3 Wireless Front End Design Details

The CC2420 is a 2.4-GHz IEEE 802.15.4 compliant RF transceiver designed for low power and low voltage wireless applications. The IEEE 802.15.4 protocol is designed for low data rate personal area networks (PANs). Sixteen communication channels are available, each of which supports a maximum data rate of 250 kbps.

The CC2420 has 33 two-byte configuration registers, 15 command strobe registers, a 128-byte transmit (TX) RAM, a 128-byte receive (RX) RAM, and an 112-byte security RAM. The TX and RX RAM can be accessed by address or accessed through two 1-byte registers, in which case the memory acts as first-in-first-out (FIFO) buffers. This report does not address writing or reading any data from the security RAM and the system does not access the TX and RX RAM as memory, only as FIFOs.

Interfacings to the registers occur over SPI, also referred to as a four-wire interface. In addition to using the SPI pins, it is also necessary to observe the signal on the FIFO, FIFOP, SFD, and CCA pins, and to drive the VREG_EN and RF_RESET pins for operation of the CC2420.

The CC2420 includes a digital direct sequence spread spectrum baseband modem providing a spreading gain of 9 dB and an effective data rate of 250 kbps. The CC2420 also provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication, and packet timing information. These features reduce the load on and allow the CC2420 to easily interface to the microcontroller hardware.

Because the CROWS demonstration is meant to be a wireless sensor network, the IEEE 802.15.4 wireless communication standard was ideal since it is specifically designed for wireless sensor PANs. The CC2420 includes several features that simplified the development of the RF capability for this project. There are few required external components needed to operate the CC2420 and the chip performs modulation, data encryption, and address recognition, and includes an onboard direct sequence spread spectrum (DSSS) modem. All these attributes can be reconfigured through software if necessary and even the RF output power is programmable with a max output of 0 dB of built in output power.

Few external components are required for the operation of the CC2420. The application circuit used in the PDHMS is shown in figure 9 and the external components shown. For more technical details on the components used, see the Chipcon CC2420 datasheet (*1*).

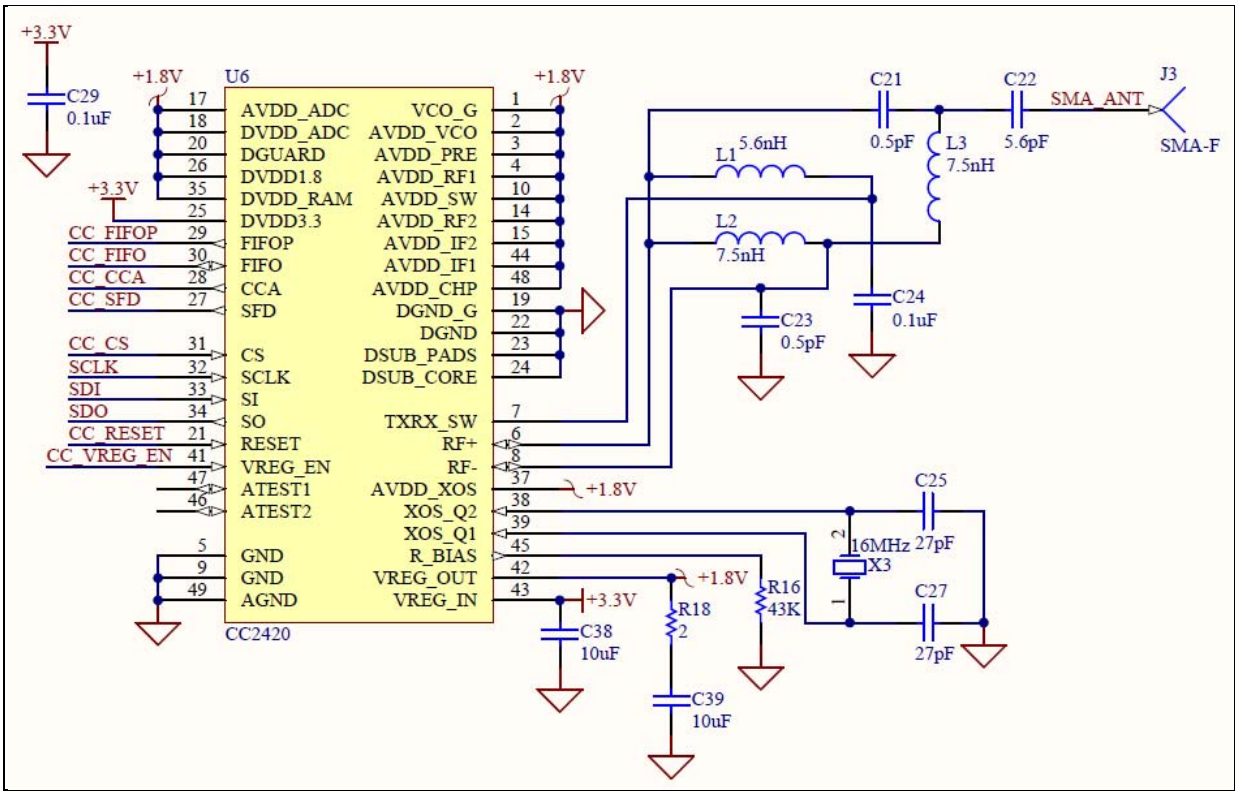


Figure 9. Typical application circuit with discrete balun for single-ended operation.

7.4 Performance Limitations of the CC2420 Transceiver

For the demonstration on the CROWS board, the 250-kbps rate was not a significant problem because we were not acquiring data at high data rates. In future redesigns, it may be necessary to go to a higher communication standard and, therefore, a different transceiver chip to increase wireless data rates.

The CC2420 is not a full duplex transceiver, which means that it cannot transmit and receive data packets simultaneously. During the development of the wireless firmware for the PDSM, we decided that when streaming large amounts of data it was ok to occasionally drop a random packet. For the purposes of the demonstration, simply streaming the data and demonstrating the overall network functionality of the PDHMS was the main priority. Therefore, although the CC2420 supports automatic acknowledgements, the firmware did not take advantage of this feature. We did not want to introduce any additional lag to the wireless communications, nor did we think the payoff for the additional time it would take to develop the firmware would add great value to our demonstration on the CROWS platform.

7.5 Wireless Networking Capabilities

For the CROWS demonstration, a star network topology was used. The primary disadvantage of a star topology is the high dependence of the system on the functioning of the central PDSM. While the failure of an individual link only results in the isolation of a single node, the failure of

the central PDSM renders the network inoperable, immediately isolating all nodes. The performance and scalability of the network also depend on the capabilities of the PDSM. Network size is limited by the number of connections that can be made to the PDSM master node, and performance for the entire network is capped by its throughput. To resolve these issues, we suggest using the CC2420 and the ZigBee stack, which also supports ad-hoc and mesh network structures with automatic route rediscovery. This type of network would be much more robust in the presence of failed nodes.

Figure 10 shows the standard IEEE 802.15.4 data packet structure for wireless communications used in the PDHMS. The structure of this data packet is what determines the order in which bytes are written to the TXFIFO for wireless transmission and read from the RXFIFO during data packet reception.

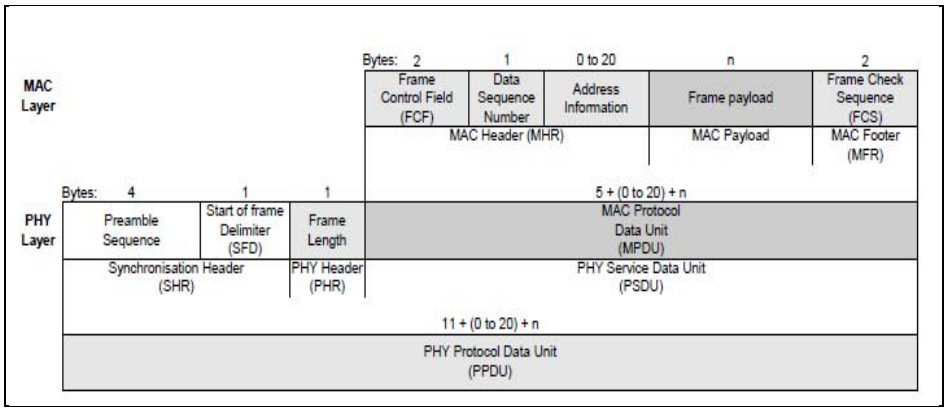


Figure 10. IEEE 802.15.4 data packet structure used in wireless PDHMS communications.

The Synchronization Header and PHY header are automatically appended onto the data packet by the CC2420 transceiver. The frame control field (FCF), data sequence number, and frame check sequence (FCS) are all defined by the firmware controlling the microcontroller. The FCF contains information such as whether acknowledgements have been turned on, whether encryption is being used, and which modes are being used. The FCF is generated based on the contents of various registers. The sequence number simply keeps track of the transmission and reception sequence of data packets between specific node addresses, which is more important when monitoring dropped packets or for automatic acknowledgements. A 2-byte FCS follows the last payload byte, as shown in figure 10. The FCS is calculated by the CC2420 over the MAC protocol data unit (MPDU), i.e., the length field is not part of the FCS. This field is automatically generated and verified by the CC2420 hardware when the AUTOCRC control bit is set in the MODEMCTRL0 control register's field. If the FCS check indicates that a data packet is corrupted, then the firmware disregards the entire packet.

The addressing information and data payload are both variable lengths. In the PDHMS application, the addressing information consists of 6 bytes: two each for the PAN ID, destination node address, and source node address. The rest of the data packet is made up of the data

payload. This payload may consist of inter-node messages, user requests, or simply sensor data being transmitted to the master node. As defined for the CROWS application, the largest acceptable data payload for wireless transmission is 111 bytes; however, all 111 bytes do not have to be used. The format of the data payload is the same as when generated for serial communications as described in section 9.10.

7.6 Real-time Clock (RTC) Design Details

The PDSM board uses the M41T93 for its RTC. The M41T93 has a SPI interface for configuration and reading the values of the RTC. Figure 11 shows the schematic of the RTC and figure 12 shows the pin connection interface between the MSP430 and the M41T93. Newer MSP430 families have integrated RTC functionality, so it is likely that the M41T93 will not exist in future designs. For more details, see the M41T93 datasheet (*7*) on its operation and capabilities.

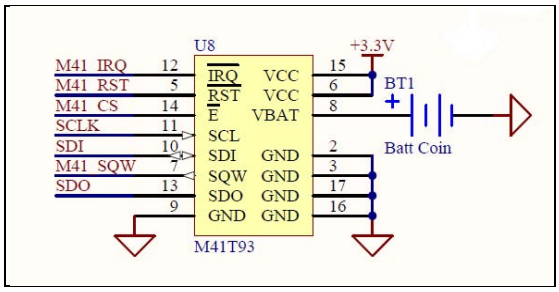


Figure 11. M41T93 schematic.

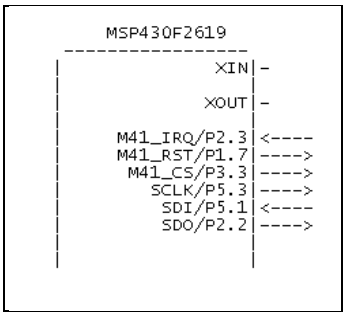


Figure 12. M41T93 to MSP430 pin connections.

7.7 PDSM Board Power Distribution Details

Figure 13 shows the power regulation circuitry for the PDSM board. It is powered by 28 V supplied at the P3 connector, with positive voltage on pin 2 and GND on pin 1. An L78L24 regulates the voltage to 24 V, which is used to power the external accelerometer circuitry. An LM9076BMA-5.0 uses the 28 V to generate 5 V, which is generally not used in the design and powers a green LED to indicate the power is on. The LM9076MBA-3.3 is used to generate 3.3 V from main power, and powers the MSP430 and most of the low power IC chips in the design.

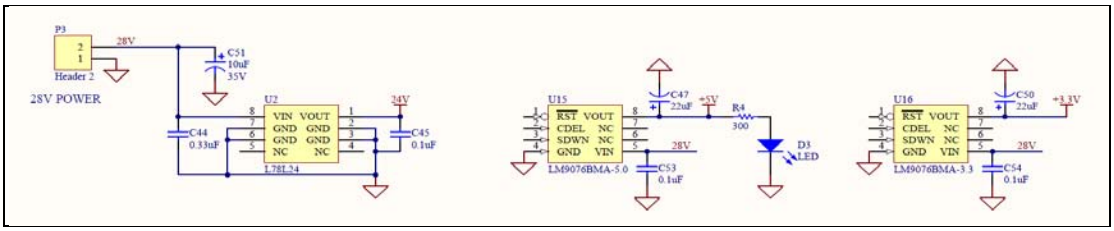


Figure 13. Power regulation circuitry.

The MSP430 and most peripherals in the PDSM design are 3.3 V or lower devices. The need for the 28-V power supply is driven by the fact that 28 V was what was available in the system in which the PDSM was to be installed, and also the external accelerometer conditioning circuitry required a 24-V power source.

8. Sensor Design Details

8.1 Thermocouple Design Details

The PDSM board's design supports connecting up to three k-type thermocouples. The hardware and firmware design of the thermocouples were primarily taken from the TI application report (*7*). The thermocouple system designed used was the ADS1241 eight-channel ADC, which supports single or differential input modes. Figure 14 shows the circuit schematic of the thermocouple design. Figure 15 shows the pin connections of the MSP430 to the thermocouple circuitry. The ADS1241 received a 1-MHZ clock from the MSP430 SMCLK signal. The ADS1241 is a SPI device controlled by the MSP430's SDO, SDI, and SCLK control signals. The ADS141 chip is enabled by driving the ADS1_CS line low. The ADS1241 is used in the differential mode to measure the voltages across the thermocouple terminals. In the design, thermocouple 1 attached across screw terminal pin 1 and pin 2; thermocouple 2 attached across screw terminal pin 3 and pin 4; and thermocouple 3 attached across screw terminal pin 5 and pin 6.

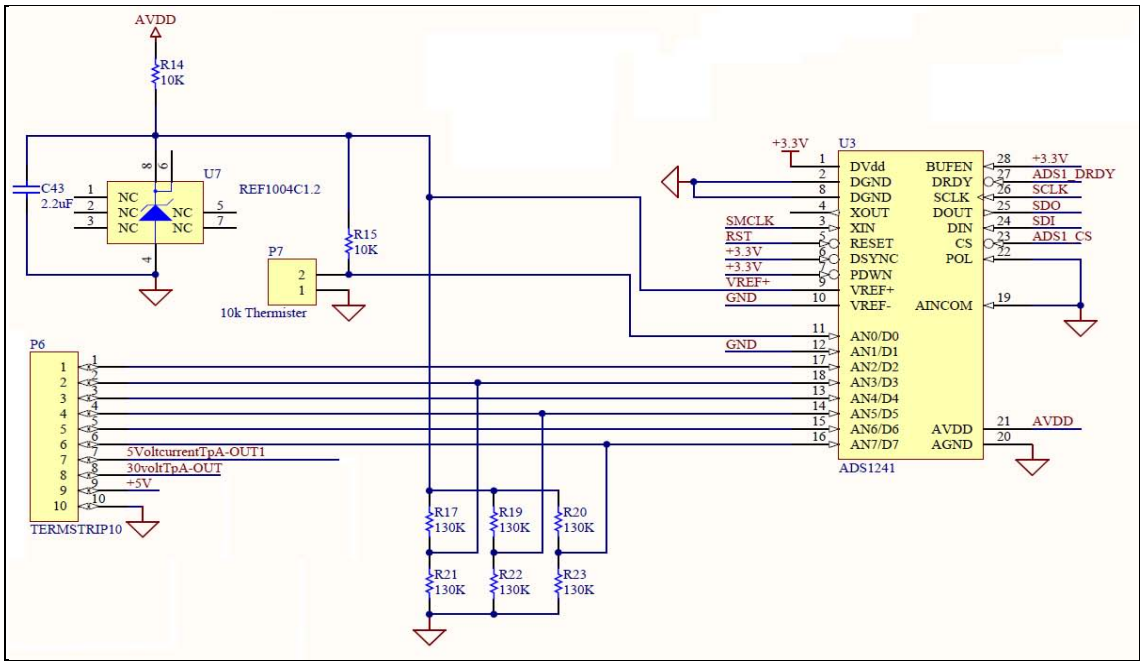


Figure 14. Thermocouple design schematic.

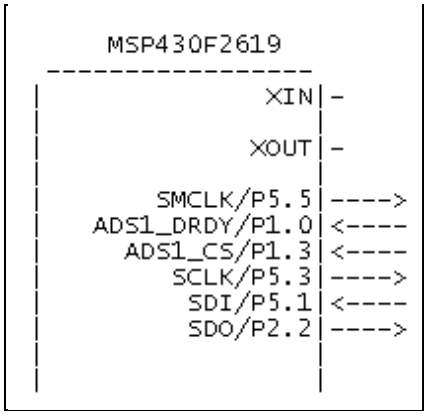


Figure 15. MSP430 pin connection to thermocouple circuits.

The differences that exist between this design and the original TI design results from the firmware of the TI design being originally written in assembly language. For this design, the firmware was completely written in C/C++ by ARL. The temperature lookup tables for this design were generated using MATLAB simulations and the datasheets of the thermisters and thermocouples. (The accompanying CD includes the MATLAB files thermister.m and thermocouple_typeK.m, which contain more details on the generation of the lookup tables used in the firmware.) In running accuracy lab tests, it was determined that the temperature measurements came to within in ±1 °F of error, once the same calibration offset was programmed across all of the boards.

8.2 Current Sensor Design Details

The current sensor is designed using the CSA-V1 Hall Effect current sensor device. The details can be found in the GMW application note (*6*). The CSA-1V devices were used in the single ended mode with its A-OUT output at 2.5 V ±2.5 V. A reading of 2.5 V implies 0 A with the device's sensitivity specified as ~44 mV/A. The circuit schematic of the CSA-1V to MSP430 interface is shown in figure 16. The A3 net label connects to pin 2 of the MSP430, which is the ADC12's A3 channel input. The maximum input of the current sensor to the MSP430's ADC should be no greater than 2.5 V per the MSP430 specification. A divide created by a two resistor divider network using two 16 k resistors and a unity gain buffer amplifier, was used to feed the A-OUT/2 to the MSP430 ADC input. A 1N5221B Zener was used for extra circuit protection.

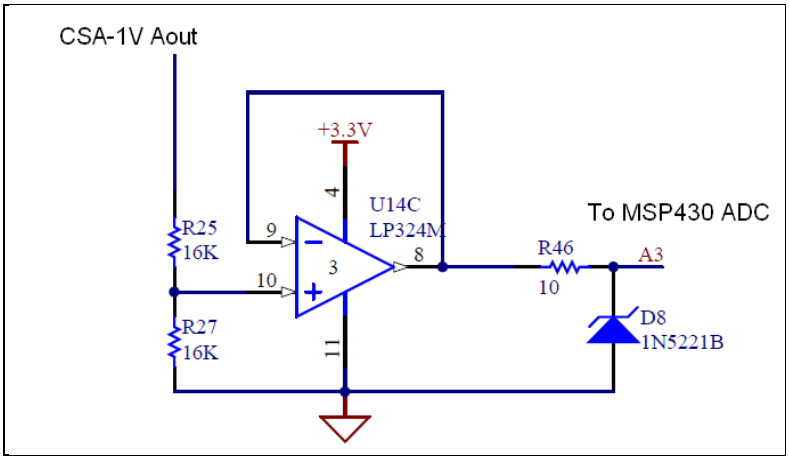


Figure 16. CSA-1V to MSP430 interface.

To convert the voltage readings at the ADC input, one can calculate current with the following analysis. In this analysis, $V_{adc} = A3$, which is the voltage input to the MSP430 for the current sensor. From the voltage divider, V_{adc} is half of A_{out} which yields equation 1:

$$V_{adc} = \frac{1}{2} A_{out} \tag{1}$$

Equation 2 is derived from how the MSP430 ADC input is configured, where V_{ref} is 2.5 V. This is an MSP430 internal voltage used as the reference for theMSP430 ADC12.

$$V_{adc} = \frac{1}{2} A_{out} \tag{2}$$

Equation 3 is obtained from the CSA-1V specification sheet (*3*):

$$I_{csa} \propto \frac{44mV}{Amp} \tag{3}$$

Assuming a linear relationship, the current reading can be computed as equation 4, where *Offset* is the zero current offset:

$$I_{csa} \cong \frac{A_{out}}{0.044} + Offset \tag{4}$$

Setting equation 1 equal to A_{out} and substituting into equation 4 yields equation 5:

$$I_{csa} \cong \frac{2 * V_{adc}}{0.044} + Offset \tag{5}$$

Substitute equation 2 into equation 5 yields equation 6:

$$I_{csa} \cong \frac{2 * (V_{ref} * ADC_{word})}{0.044 * 4095} + Offset \tag{6}$$

Plugging in values yields equation 7:

$$I_{csa} \cong 0.0278 * ADC_{word} + Offset \tag{7}$$

Now assuming that when $I = 0$, $ADC_{word} = 2048$, we solve equation 7 and get *Offset* = 56.9344. Equation 7 can then be written as

$$I_{csa} \cong 0.0278 * ADC_{word} - 56.9344 \tag{8}$$

Equation 8 approximates converting the ADC_{word} readings to amperes. Plugging the value for ADC_{word} into equation 8 to the limits of 0 and 4095 suggests the limits of the current sensor is ±56.9344 A. To verify the accuracy of equation 8, three different CSA-1V sensors were connected to the same PDSM board to take measurements of a circuit consisting of a variable power supply connected across the terminals of a high power resistor. The power supply voltage was varied to generate currents from 0 A up to 12 A, in increments of 0.5 A, and the readings were taken using a current meter. Also, the corresponding readings of the ADC_{word} on the CSA-1V current sensor were taken by the MSP430. Equations 9–11 were computed based on the three data sets, and these represent the correct scaling functions in scaling the ADC_{word} readings to current for each of the CSA-1V sensors used in the CROWS demonstration:

$$\text{CSA-1V \#1 : } I = 0.0309 * ADC_{word} - 62.868 \tag{9}$$

$$\text{CSA-1V \#2 : } I = 0.0301 * ADC_{word} - 61.113 \tag{10}$$

$$\text{CSA-1V \#3 : } I = 0.0296 * ADC_{word} - 60.614 \tag{11}$$

As shown above, all CSA-1V devices have different intersects and slopes, and all differ from equation 8. These measurements suggest the need to calibrate each sensor for scaling accuracy. This requirement is not practical if large numbers of these types of sensors are used. This would require that all PDSM nodes be programmed uniquely with a scaling equation as above, and

calibration would be required every time a different CSA-1V sensor is used. This situation also raises the concern that there is a logistics requirement in knowing which current sensor is attached to a given PDSM board. Because of this, it may be necessary to investigate another type of current sensor that will not require a calibration procedure for each sensor.

8.3 Voltage Sensor Design Details

The voltage sensor was implemented using a resistor voltage divider fed to the input of a LP324M operational amplifier wired as a unity gain amplifier, as shown in figure 17. Figure 17 shows the voltage test point, Vs, on pin 8 of the terminal strip connector. That connector feeds across the voltage divider network, creating voltage Vi at pin 5 of the operational amplifier. The output of the operational amplifier, pin 7, feeds A4 of the MSP430 through a 10-ohm resistor, which is there to prevent circuit oscillations. Because of the configuration of the circuit, approximately the same voltage level Vi is assumed to be at pin 5, pin 7, and across the D7 diode. A 1N5221B Zener was used for circuit protection. The input impedance of the MSP430 ADC is nominally 2 K ohms, so there is negligible voltage drop across the 20-ohm resistor. For measurements in the CROWS, we wanted to measure at most 30 V from the CROWS power line. With R24 and R26 having values of 33 K and 2 K ohms, respectively, the following circuit analysis shows that the voltage sensor can safely measure 0 to 43.75 V on its input when fed to a MSP430 ADC12 input.

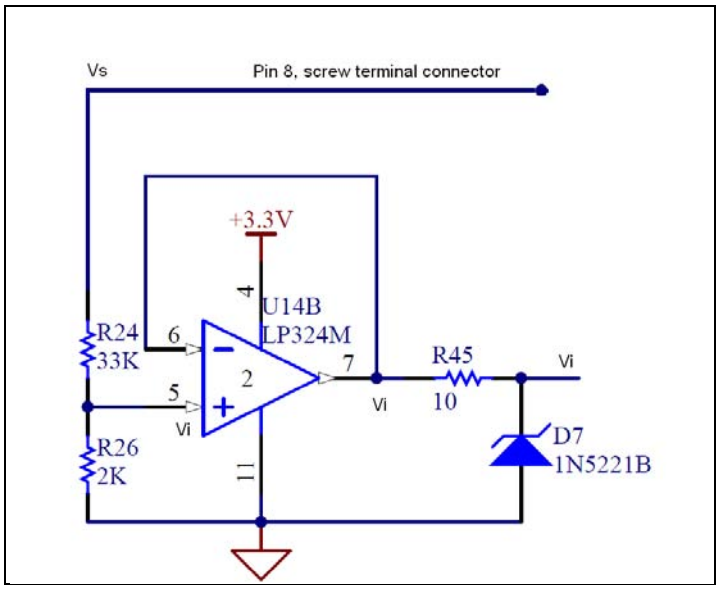


Figure 17. Voltage sensor implementation.

Applying the resistor voltage divider circuit yields equations 12 and 13:

$$V_i = \frac{2 * V_s}{33 + 2} = \frac{V_s}{17.5} \tag{12}$$

$$V_s = 17.5 * V_i \tag{13}$$

Assuming the reference voltage for the MSP430 ADC12 is set to the internal reference of 2.5 V (the maximum voltage that should reach the ADC12), the maximum voltage that should be applied to the voltage point is shown in equation 14:

$$V_{s\max} = 17.5 * (2.5) = 43.75 \text{ volts.} \tag{14}$$

Equation 14 gives the maximum voltage that should be applied across the voltage test point to ground. Exceeding this voltage can damage the MSP430 processor.

The procedure to scale the ADC12's data word readings to voltage follows. The ADC12 voltage as a function of the ADC word value is

$$V_i = V_{adc} = \frac{V_{ref} * ADC_{word}}{4095} = \frac{2.5 * ADC_{word}}{4095} \tag{15}$$

Substituting equation 1 into equation 4 yields equation 5,

$$V_s = \frac{43.75 * ADC_{word}}{4095} \tag{16}$$

Equation 16 computes the test point voltage reading as a function of the reading taken on the ADC12 input.

8.4 Onboard Accelerometer Design Details

The PDSM has a MMA7260Q accelerometer onboard. The perceived application for this accelerometer is to allow orientation measurements of the equipment being monitored, which could be useful if one needs to correlate other sensor measurements with the platform orientation or measure vibrations that the PDSM is exposed to when mounted on a platform. The Freescale Semiconductors MMA7260QT Rev 5 technical datasheet (*2*) provides the technical details. Figure 18 shows the connections of the accelerometer interfaced to the MSP430. The accelerometer readings are taken from the MSP430's ADC12 A0, A1, and A2 channels for each of the three axis inputs. The accelerometer has four sensitivity levels of 1.5G, 2G, 4G, and 6G controlled by pins P4.5, P4.4, and P4.3 of the MSP430.

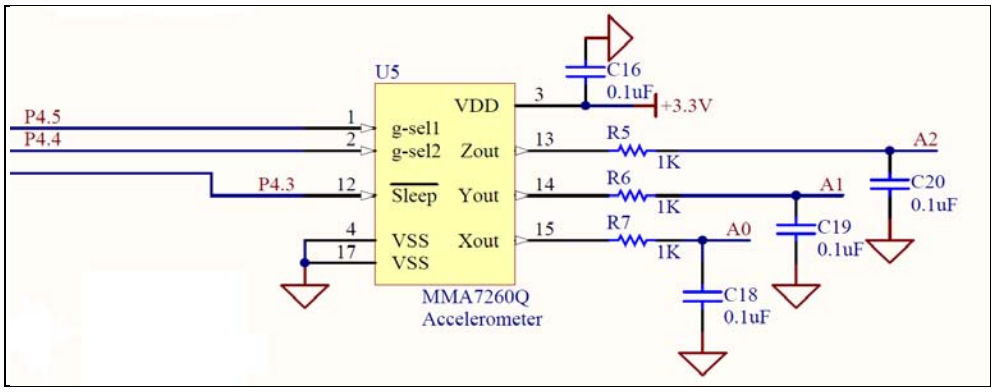


Figure 18. MMA7260Q accelerometer connections to the MSP430.

When taking measurements on these inputs, the MSP430's ADC12 reference is set to a Vcc of 2.5 V.

Equation 17 gives the voltage across the inputs of the three axes:

$$V_{adc} = \frac{V_{ref}\, ADC_{word}}{4095} \tag{17}$$

From the MMA7260Q data sheet's static acceleration specifications, one can derive the angular positions of the device by computing the linear equations from the points in table 3.

Table 3. MMA7260Q static acceleration voltage verses angle.

Voltage	**Angle (°)**
2.45	0
0.85	180

Here, angle is the angle of a given axis relative to the direction of the Earth's gravity and voltage is the reading taken by the ADC12. From this, equation 18 gives the static angle in degrees as a function of the voltage readings.

$$Angle = -112.5V_{adc} + 275.63 \tag{18}$$

where the valid voltages as in the table range from 0.85 to 2.45 V.

Substituting equation 17 into equation 18 with V_{ref} = 2.5 V yields the angle calculation as a function of the ADC12 readings:

$$Angle = -0.687 * ADC_{word} + 275.63 \tag{19}$$

Equation 19 converts the binary ADC12 readings to angular values for each axis. All three readings can be used to determine the exact orientation of the PDSM within the platform.

8.5 External Accelerometer Design Details

The Vibra-Metrics M3000 tri-axial accelerometer is shown in figure 19. The accelerometer (part number 9353354) is a 10-mV/G accelerometer with a dynamic range of ±500G. It is a piezoelectric low impedance transducer that requires 15–30 V of DC input to power each axis. A DC bias of 7 V is generated when properly conditioned. The Vibra-Metrics Accelerometer User's Manual, Rev. 2, June, 2004, Part #9350-1000 (*8*) provides guidance on interfacing the M3000.

Figure 19. M3000 Vibra-Metrics external accelerometer.

The conditioning circuitry of figure 20 is designed to supply the proper current to the M3000 accelerometer. A LM334D current source with a 1N457 temperature compensating diode was used to bias each axis on the M3000. The National App Notes, March 20005, LM134/LM234/LM334 (*5*) provides complete details of this design circuit. Figure 21 shows the actual circuit.

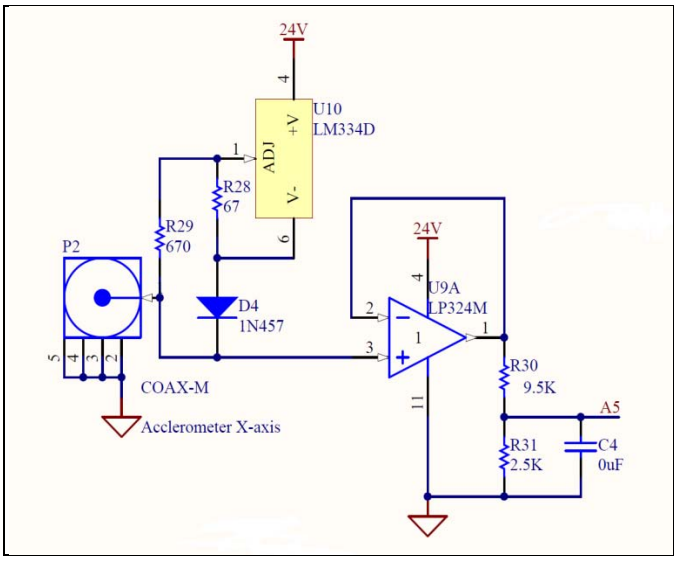


Figure 20. M300 *x*-axis conditioning circuitry and connection to MSP430.

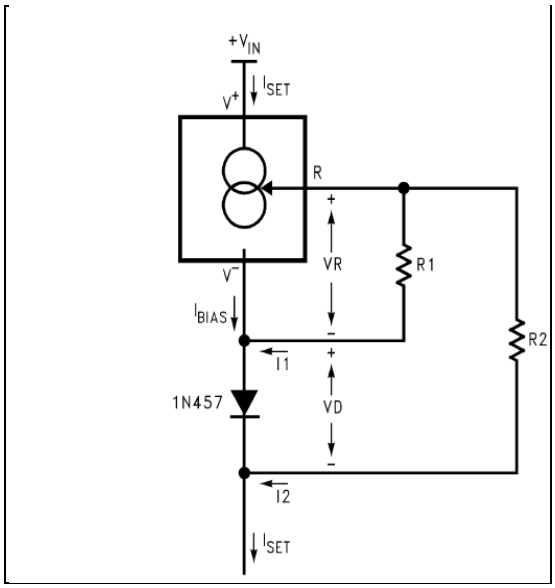


Figure 21. M3000 external accelerometer conditioning circuit.

Using the LM334D application notes guidelines, the general equations for the circuit are $I_{SET} \approx 0.123V/R1$ and $R2/R1 \approx 10$. Setting I_{SET} =2ma, results in R1 = 67 ohms and R2= 670 ohms. These values were used in the design with reasonable results.

To use the M3000, the specs require that it is driven with a nominal current source of 2 mA. The M3000 specification sheet suggests a bias current from 1–6 mA. The Vibra-Metrics application notes suggest biasing the current to 2 mA, so this was used as our design goal. Connecting a 28-V source, the 24-V regulator regulated to about 23.8 V. The measured bias voltage returned from connected M3000 accelerometer (s/n 3069) was about 6.5 V. This voltage appeared to be within a reasonable range of the expected 7 V.

The accelerometer G-calculation is performed based on the M3000 specification of a voltage variation around the DC bias voltage of 10 mV/G. The ADC sensitivity is 2.5 V/4096 = 0.61 mV/bit. Table 4 shows measurements made with a voltmeter across the output pins of the M3000 accelerometer while the circuitry was fed to the ADC of the MSP430. All of the axes measurements are reasonably close to the expected 7 V. Differences are likely due to error tolerances in the components used and actual regulated voltage levels. These static readings have no significance in the actual dynamic readings; however, they do suggest the need to perform a calibration procedure to establish 0G acceleration offsets so that these values can be subtracted from dynamic measurements.

Table 4. Voltmeter reading across LM334D voltage M3000 orientation.

	X pin1	**Y pin 7**	**Z pin 8**
***x*-vertical**	6.37	6.41	6.46
***y*-vertical**	6.35	6.40	6.44
***z*-vertical**	6.38	6.41	6.45

8.6 Resistor Divider Network Computations for Accelerometer Op-amp

To minimize circuit loading effects on the M3000's bias current, a high input impedance unity gain buffer operational amplifier was used after the current source, as shown in figure 20, to feed the accelerometer output to the MSP430. The operational amplifier's rail voltage is set to 24 V. The M3000 accelerometer is specified to output ±500G with 10 mV/G. This setting implies a voltage swing of 500G × 10 mV/G = ±5 V on the output of the M3000. Further, this suggests that the operational amplifier voltage output, which has been measured at ~6.4 V with no acceleration, can swing ±5 V around that level giving a maximum possible swing from 1.4 to 11.4 V. The operational amplifier output must be able to handle these levels. The LP324 has a low level worse-case output voltage of 1 V, well below 1.4 V, and a high level worse-case voltage of 24–1.9 = 22 V. Thus, the LP324 can accommodate the full swing level of the accelerometer.

8.6.1 Computation of Resistors

In determining appropriate resistor values for feeding the MSP430 ADC12, TI Application Report SLAA148 (*4*) was referenced. On the output of the buffer, a voltage divider is needed to condition the output to within the MSP430 voltage range and selection of the resistor. The circuit feeding the ADCs from the M3000 is a unity gain op-amp fed to the resistor divider network. To calculate the required resistors values, the Thevenin equivalent of the op-amp resistor divider network feeding the ADC12 is

$$V_s = \frac{R_2 * V_{op-amp}}{(R_1 + R_2)}, \tag{20}$$

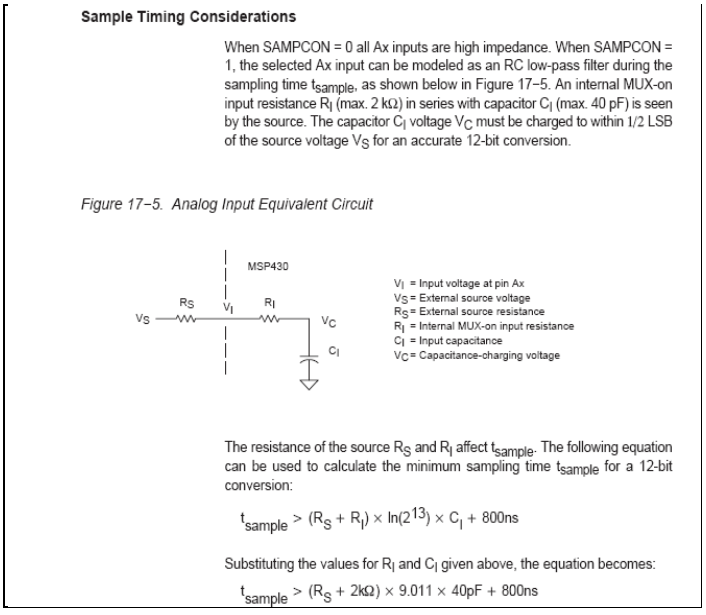
and the Thevenin resistance is

$$R_s = \frac{R_1 R_2}{R_1 + R_2} \tag{21}$$

Arbitrarily setting R_1 = 9.5 K and R_2 = 2.5 K yields V_s = 2.357 with V_{op-amp} = 11.5 V and yields 0.2197 V when V_{op-amp} = 1.4 V. The original operational amplifier used in this design was the LP324, where its lower voltage limit restricted to 1 V, which effectively limited the dynamic range of the accelerometer sensor. The latest design used the LM324, which had a lower limit of 0 V, thus allowing a wider dynamic range. It was observed in measurements that although the LM324 did provide a wider dynamic range, it was much noisier than the LP324. Selecting the appropriate amplifier requires further investigation.

8.6.2 Sampling Rate Estimate

To estimate the sampling rate of the ADC12, figure 22 was taken from TI MSP430 application notes and shows the equivalent circuit for timing considerations of the ADC12. Figure 22 shows the ADC12 input with voltage source V_s source resistance R_s, and internal resistance R_i with typical value assumed to be 2 K. From equation 21, R_s is computed to be approximately 2 K, and t_{sample} as computed in figure 22 must be greater than 2.24 µs. Although this suggests a sampling rate of 446 kHz, sampling is further restricted by the ADC12 maximum sample rating of about 200 ksps. Laboratory measurements were shown to give reasonable sample rates of about 100 ksps or less, but this will vary depending on the input load to the ADC12.



Sample Timing Considerations

When SAMPCON = 0 all Ax inputs are high impedance. When SAMPCON = 1, the selected Ax input can be modeled as an RC low-pass filter during the sampling time t_{sample}, as shown below in Figure 17−5. An internal MUX-on input resistance R_I (max. 2 kΩ) in series with capacitor C_I (max. 40 pF) is seen by the source. The capacitor C_I voltage V_C must be charged to within 1/2 LSB of the source voltage V_S for an accurate 12-bit conversion.

Figure 17−5. Analog Input Equivalent Circuit

The resistance of the source R_S and R_I affect t_{sample}. The following equation can be used to calculate the minimum sampling time t_{sample} for a 12-bit conversion:

$$t_{sample} > (R_S + R_I) \times \ln(2^{13}) \times C_I + 800ns$$

Substituting the values for R_I and C_I given above, the equation becomes:

$$t_{sample} > (R_S + 2k\Omega) \times 9.011 \times 40pF + 800ns$$

Figure 22. Schematic sample timing.

8.7 SD/MMC Card Design Details

The schematic of the SD card connection to the MSP430 is shown in figure 23. The SD card implementation of the SPI protocol for communications between the MSP430 and the SD to MSP430 pin connections are shown in figure 24. On pin 6, a 2 K pull-up resistor is used to detect when the memory card is inserted into the SD Card Hirose connector. Inserting the memory card into the connector causes the chip detecting a voltage level on pin 6, SD1_CD, to

be pulled to ground. The MSP430 firmware is programmed to detect ground level to confirm SD card insertion. The serial data input is connected to pin 2, serial data output is connected to pin 7, and the serial clock SCLK is connected to pin 5 of the SD card. The basis of the firmware and hardware in this design was derived from TI Application Report, SLAA281A–November 2005–Revised January 2006 (*9*).

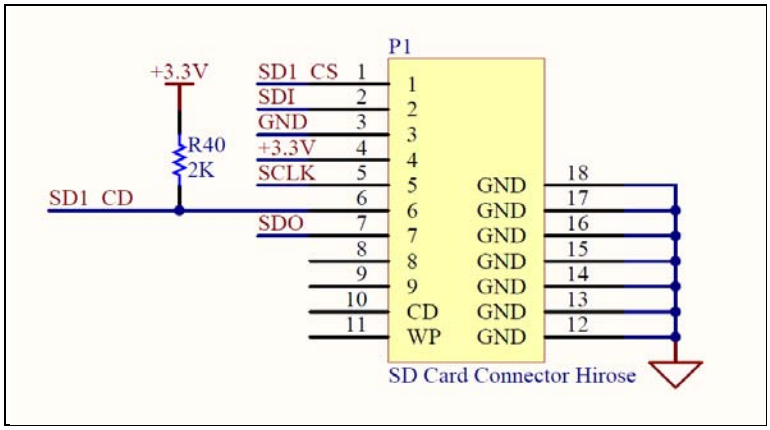


Figure 23. SD/MMC card schematic.

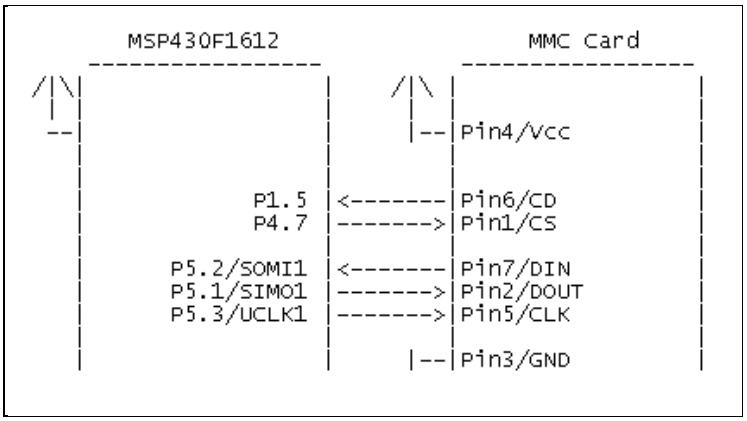


Figure 24. MSP430 to SD/MMC interface.

8.8 MSP430 Clock Use and Distribution Design Details

This section describes the use of the MSP430 clocks and the clock source, defining which peripherals use which clocks of the MSP430 and the desired clock rate settings of each. Given the difference in clock speeds for the various peripherals, it is important to keep in mind the settings of these clocks and their sources. Performance of the peripherals is affected by the various MSP430 clock settings. Care must be taken in the firmware to manage these clock rates. Table 5 is presented to make the developer aware of the need to pay close attention to the clock settings and how they impact the system. The clock settings are primarily dictated by how fast data must move in the system, clock specifications of the peripheral devices, and system power requirements.

Table 5. Overview of the clocks embedded onboard the MSP430 chip and the corresponding clock sources.

MSP430 Clock	Peripheral	Speed	Clock Source	Comments
MCLK	MP430 CPU	8 MHz (16 MHz)	XT2 crystal	A CPU clock. Preferred to run at this rate to maximize data processing, data transfers, storage rate, and communications.
MCLK or ADC12OSC	ADC12	8 MHz (16 MHz) or nominal 5 MHz with ADC12OSC	XT2 crystal	The actual rates affect sample and hold. Setup times are defined by the ADC12 registers. Review these carefully in the msp430 documentation. This clock rate is not the same as the sample rate of ADC12. The ADC12 sample rate is dictated by sample and hold setup times and the Timer A1 interrupt rate as used in the firmware. See msp430 documentation and firmware for more details.
SMCLK	Timer A1	1 MHz	MSP430 internal digitally controlled oscillator (DCO)	Timer A1 is used for the overall sampling rate of ADC12, taking into consideration setup/hold/conversion times as discussed above.
SMCLK	UART	1 MHz	MSP430 internal DCO	The UART requires a fixed rate clock to get a 115200-baud rate. The MSP430 and GUI are presently hardwired to 115200 baud.
SMCLK	ADS1240	1 MHz	MSP430 internal DCO	The ADS1240 clock rate cannot be greater than 4 MHZ; however, this clock can be locked at the lower 1 MHZ, since we are sampling at such a low clock rate. Specs indicate that ADS1240 clock minimum is 1 MHZ.
SMCLK	I2C	1 MHz	MSP430 internal DCO	Clock source selection is done in the I2C master initialization code. It is presently set to SMCLK, which is set to 1 MHZ on the DCO.

9. Firmware System Level Design

This section describes the firmware design of the PDSM. Figure 25 is a block diagram of the architecture of the network communication of the PDHMS. The communications for the PDSM boards uses a common approach where all communications and system behavior is message driven. With the message driven paradigm, a single master (client) and multiple slave (servers) topology is used in the form of a star network (as shown in figure 16). The master is typically connected to the PDCS computer via a USB port. The PDCS runs the system command and control GUI. Through the GUI, the user can issue commands to the master to configure the master itself and/or all of the slave nodes in the system. The master is the connection point between the PDCS and all slave nodes in the system, thus the master acts as a communications broker in the architecture. The master can issue commands such as making status requests of each node, and can send configuration commands to each node and data acquisition commands to the nodes. Each master and slave pair has a unique 3-bit address identification number that is

configured by setting the appropriate jumpers. The 3-bit address limits the number of nodes in the system to eight. However, with minimal design change, the number of nodes in the system can be increased to whatever is required. The master node must always be connected to the PDCS, and its address identification number must always be set to zero (000). The slave addresses must be set to settings from 1 through 7 (001–111). To avoid communications conflicts in the network, care must be taken to ensure the address identification numbers of each PDSM is unique. These node address settings are used by the USB/USART, wireless, and I2C communications mediums in the system.

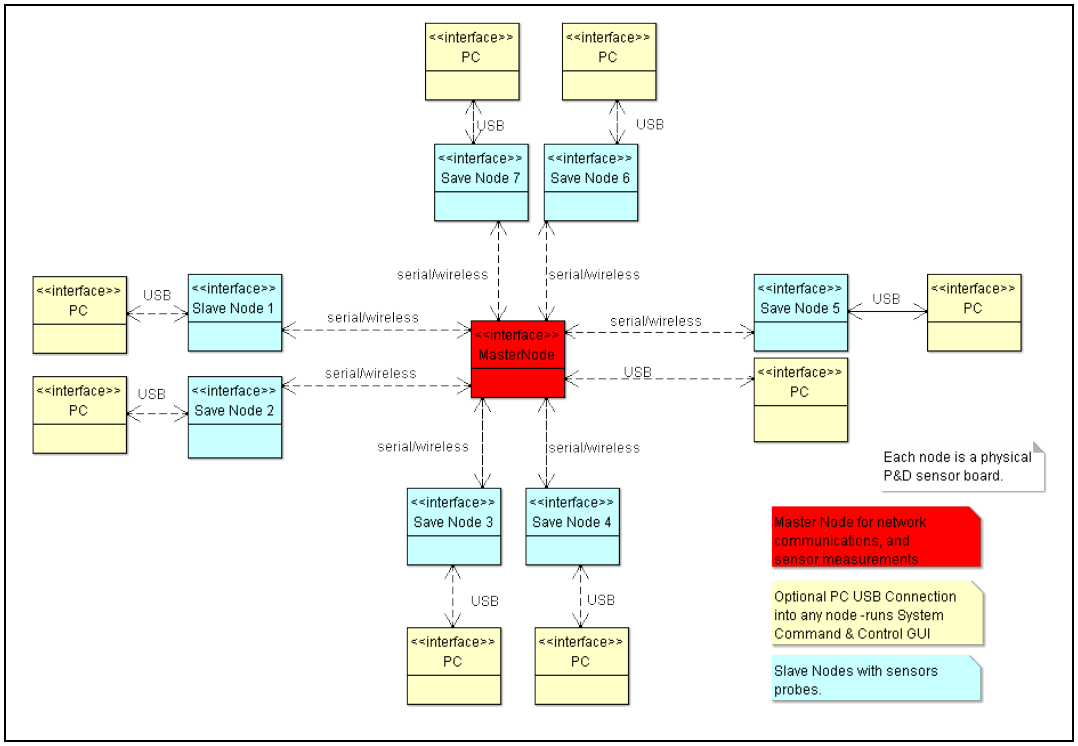


Figure 25. Inter-node star network communication hierarchy.

9.1 Setting PDSM Jumpers

Figure 26 shows the circuit schematic and table 6 shows the node addresses versus PDSM jumper settings. We used JMP0, JMP1, and JMP2 to set the PDSM address identification numbers, which corresponds to P3.0, P5.6, and P5.7 of the MSP430. P3.0, P5.6, and P5.7 are tied to pull-up resistors via 2-K resistors. When attaching jumpers JMP0, JMP1, or JMP2, the corresponding pin gets pulled to ground. The firmware is written to use the inverse logic levels of the lines so that setting the jumpers gives addresses that are more natural to the user.

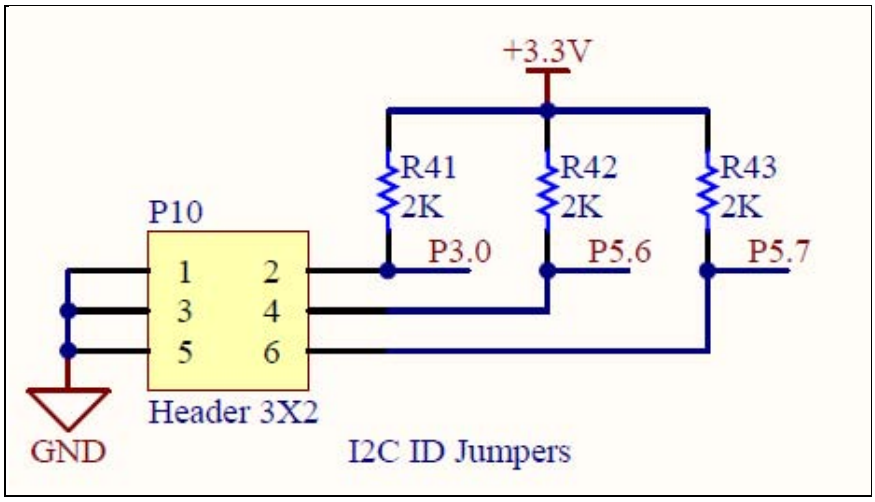


Figure 26. PDSM jumper schematic.

Table 6. Node address versus jumper settings.

JMP2	JMP1	JMP0	Address	Node Name
off	off	off	0x0	Master
off	off	on	0x1	Slave1
off	on	off	0x2	Slave2
off	on	on	0x3	Slave3
on	off	off	0x4	Slave4
on	off	on	0x5	Slave5
on	on	off	0x6	Slave6
on	on	on	0x7	Slave7

The slave boards respond to messages sent from the master through the various communication mediums. Each communication medium supports access to all defined commands within the system. When a slave or master receives a command request on a particular medium, it always responds on the same medium on which the request arrived.

9.2 Communication Network Design Decisions and Limitations

Each PDSM board has a USB connector. The connector is used to allow the user to issue commands to the board through the PDHMS GUI if necessary. All nodes have the exact same copy of firmware running on them. The node with jumper ID zero behaves as a master node and the other jumper IDs behave as slaves. To the end user, this means that connecting a PDCS into the USB of the master gives the user remote access to all nodes in the network through the command structure. However, connecting a PDCS into the USB of a slave only gives the user access to control the slave to which the PDCS is physically connected. The user cannot reliably communicate from a slave address ID out to another node in the network with the PDCS connected to the USB connection of a slave node. The current firmware does not support this ability. We implemented the system in this manner to limit the communications firmware design complexity and allow the user a little more flexibility in debug and development. A later version

should probably allow a PDCS to connect to any slave via USB and communicate to all nodes in the network, effectively allowing any slave node to serve as a master node. This capacity should be much easier to implement when using an RTOS.

The USB interface uses pins 33 and 32 of the MSP430. This makes use of the MSP430 interrupt vectors USCIAB0TX_VECTOR and USCIAB0RX_VECTOR for transmit and receive USART operations. The I2C interface makes use of pins 29 and 30 for communications, using the interrupt vectors USCIAB0TX_VECTOR and USCIAB0RX_VECTOR for transmit and receive operations. The interrupt handler must process interrupts for multiple communications channels. Interrupt flag registers must then be monitored to determine the actual source of the interrupt to process the interrupts correctly. This process increases the complexity of software integration between differing communications mediums, which is one of the reasons we created the **int-lib** to force these commonalities into one location in the software.

9.3 Medium Communications

To perform communications through IEEE 802.15.4, I2C, and USB, we developed a high level application layer of function calls. These calls are required to isolate the general P&D application software from the underlying details of the communications mediums. In this process, *receiveMsg()* pseudo code handles incoming messages originating from IEEE 802.15.4, I2C, or USB, and *sendMsg()* allows the PDSMs to send messages to the desired destination:

- *receiveMsg()*—All receive communications are interrupt driven. When a received data communications interrupt occurs, the *receiveMsg()* function is called to handle the message. Depending on the interrupt source, *receiveMsg()* calls the appropriate communications device driver to receive the incoming message. Upon return from *receiveMsg(),* the parameters of the function contain the message source, the message command, the length of the data, and the data placed in the data buffer. The valid values of a message source are dI2C, dZigBee, and dUSB. These values tell the slave where to respond: Cmd indicates the command the slave must perform, dataLen shows the length of the data, and dataBuff contains the received data. The following is an example of *receiveMsg()* code:

```
void receiveMsg(unsigned short *msgSrcAdrr, unsigned short *medium,
    unsigned *cmd, unsigned *dataLen,  char *dataBuff);
```

- *sendMsg()*—All communications messages sent by either a master or slave are done through the *sendMsg()* function call. The communications channel used to send the message is msgDst. The valid values of a message source are dI2C, dZigBee, and dUSB.

  Cmd indicates the message command, dataLen shows the length of the data to be sent, and dataBuff contains the data to send. A value of true is returned if the send is successful; otherwise, false is returned upon failure. The following is an example of *sendMsg()* code:

```
bool sendMsg(unsigned short msgSrcAdrr, unsigned short medium,
                    unsigned cmd, unsigned dataLen,  char *dataBuff);
```

9.4 Message Bus Architecture Design

Figure 27 shows the general mechanism for inter-processor communications within the PDHMS. Although this example shows communications from the GUI to one slave node, this mechanism is used to communicate with all nodes in the system. Each message sent on the message bus must have a message header. The message header defines the originating source of the message, the destination node of the message, and the gateway to be used to pass the message from source to destination. The source, destination, and gateway are all defined by two parameters: medium and address. When a node initiates communications on the message bus, it must fill in this header information correctly for the message to be sent to the proper destination and for a potential reply message to be received back to it.

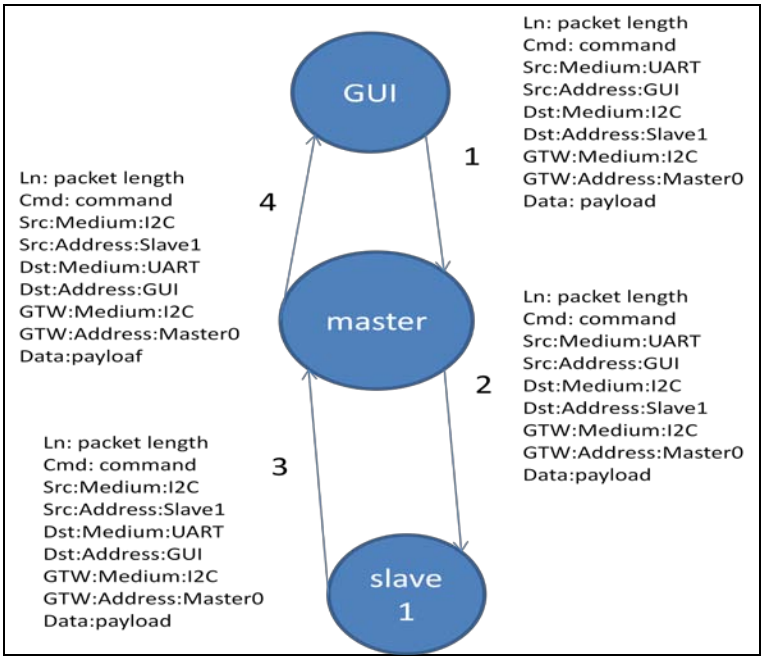


Figure 27. Message bus architecture.

In the example shown in figure 27, the GUI wants to send a message to slave node 1, and slave node 1 sends a message back to the GUI. This process is performed using the following 4 steps:

- Step 1: The GUI node fills in the header as indicated by “1” in figure 27. The message from the GUI always moves across the UART (USB) connection. The GUI configures the source medium as UART and the source address as GUI. The GUI node also fills in the destination medium as I2C and destination address as slave1. In the present system, the gateway is always configured to be the masterNode0 (address 0) and the medium in this example (what the gateway uses to talk to the slave) is configured as I2C. The GUI sends a message with this header information to the master node, which is always the gateway.

- Step 2: Once the master node receives the message sent from the GUI in step 1, its job is to determine if the message is for the master node or if the message should be forwarded to a destination node. If the message is intended for the master node, the master node processes the message according to the command set. In this example, however, the master node is required to forward the message to slave node 1 across the I2C bus as indicated by the destination setting in the message as sent out by the GUI. So, the master forwards the message out to the I2C bus to slave 1 with the original information unmodified.
- Step 3: The slave 1 receives the message and processes the message according to the command set. If the slave is required to reply back to the originating node of the message it has just received, the slave uses the header information to determine where to send a reply message. In this example, slave 1 sets the source medium as I2C (based on the medium used by the message originator, in this case, the GUI) and the source destination as slave1. The slave sets the source medium to be whatever the original source medium was from. Using the same medium guarantees that the message will get back to the GUI since it is communicating on the same channel as the message originated. Since this is a slave node (slave 1), it uses the gateway medium and address information to send a message back to the GUI. In this example, slave 1 sends a message using the gateway medium as I2C and the gateway address as master0.
- Step 4: Upon receiving the message from the slave, the master node again determines if the message is for itself (and processes it if it is) or forwards the message onto the destination node. In this case, the master node forwards the message unchanged to the GUI using the destination medium (I2C) and address (GUI) as defined in the message. This design allows slaves to cross communicate as required.

9.5 Communications Message Format

What follows is pseudo code of what the actual message formats are in the system. All data types are little-endian, which is derived from the MSP430 architecture.

Every message sent or received in the network is communicated in the form of one or more message packets. The number of packets must form a complete message as defined in the msgPacket structure. The msgPacket consists of a message header, optionally followed by a data payload.

The packet msgHeader has several fields. The first 2 bytes of the header contain the hexadecimal synchronization codes 0xaa and 0x55. These values must be at the beginning of packet header and are used for packet data integrity checks. These values are always checked on the reception of a packet, and if they are not there, the complete packet is ignored. This check is done as a means to detect dropped or invalid packet data. The length field is used to determine the length of the complete packet, which includes the byte length of the packet header and the data payload. Although the length field is a 2-byte unsigned short integer, the maximum value of length is

restricted to greater or less than the value of MAX_PACKET_LENGTH_BYTES. The command field is a 2-byte short integer, which defines the command transmitted by the message. The valid values of the command field are defined by the enumerated type PdCommandSet.

The packet data payload is optional, because some messages do not have a data payload, but only a command. Each message packet size is limited to the size of the message header plus the size of the maximum allowed data payload. The design defines the maximum packet data payload to be MAX_MSG_DATA_LENGTH_BYTES. The maximum size of the packet data payload is dictated by various aspects of the hardware, such as the available RAM memory of the MSP430 microcontroller or the largest byte size a message can be sent through a given communications medium (i.e., through the wireless CC2420 chip, as was the case for this design). The total packets field defines the total number of packets that make up a complete message. The receiver of multiple message packets is required to reassemble the packet message before processing the message. The packet number field defines which packet of the total packets is being sent, and this value counts from one to the total number of packets. The source field defines the source node identification and medium. This information allows the receiver of a message to reply back to the originator, if desired. The destination field is the destination node ID and medium. The gateway field is always the master's node address and medium. All slaves communicate through the master gateway back to the GUI.

For the network system to operate properly, a critical point to consider in this design is that all nodes communicating in the system must adhere to the same message command structure. All nodes must be programmed with the same command tables for proper command processing. If the command table on the GUI software is updated, all nodes in the network must be reprogrammed with the same command table as the GUI. Conversely, if the command table on the MSP430 is modified, the GUI code's command tables must be updated to the same values.

A complete message is made up of multiple packets. The maximum number of packets for a complete message is defined by the totalPackets field, which has size of "char." "Char" limits the maximum number of packets per message to 255. Furthermore, for the present design, the maximum number of bytes allowed per data payload is defined by MAX_MSG_DATA_LENGTH_BYTES, which is set to 80 bytes. This setting implies that the total data length of a complete message in the network can be no greater than $80 \times 255 = 20400$ bytes. These values can be adjusted depending on the need of the PDHMS, but these restrictions are driven primarily by the limited RAM in the MSP430. If messages greater than this are required, there are several options available. One could design a higher level message structure that could be imposed on the interpretation of the data, use a bigger data size for totalPackets, or consider using a MSP430 with a larger RAM that would allow increasing the data payload size, among others.

As a design rule, slaves should not be sent messages of sizes greater than one packet. This rule is due to the limited RAM space that slave nodes have to work with. To date, our design has been

able to achieve this requirement. In contrast, slaves must be able to send messages composed of multiple packets, for instance, when slaves are commanded to send acquired data that span multiple packets due to the size of the number of samples during a sensor acquisition.

The format of the message structures described previously is as follows:

```
typedef struct
{
        msgHeader hdr;
        char  *data; //[MAX_MSG_DATA_LENGTH_BYTES]; new
} msgPacket;


typedef struct
{
        unsigned char haa;
        unsigned char h55;
        unsigned short ln; //length of this packet
        unsigned short cmd;      // command
        unsigned char totalPackets;// total number of packets for a complete message,
                                   //val is 1 or more
        unsigned char packetNumber;// this packet number, 1 up to totalPackets
        ChannelType src;
        ChannelType dst;
        ChannelType gtwy; //gateway, generally the master node attached to the usb gui.
} msgHeader;


typedef struct
{
        unsigned short medium; //use enum commsmedium
        unsigned short node_address; //use enum pdnodeaddr
} ChannelType;
```

9.6 Pseudo Code, Node Message Processing

This section describes the design behavior of the PDSM master and slave boards. The primary function of the master PDSM is to transmit configuration and status commands between the PDCS computer and PDSM slave boards. The master's job is to issue the desired commands to the slaves according to the defined command structure described previously. The primary task of the slave nodes is to acquire data on the sensors they are configured to monitor and pass any requested information back to the PDCS. Although the master and slave nodes conceptually have different tasks, they both run the same firmware. This design decision was made to simplify firmware development; thus, only one copy of firmware is required for programming all the nodes. As previously mentioned, the node address identification jumpers dictate if a node behaves as a master or a slave. At the user API programming level, whether a master or a slave, the nodes perform the same type of message processing operations. The pseudo code of the behavior nodes is as follows.

The network was designed so that the only master issues master node commands to the slaves. A master node can also issue slave-related commands, because it can act as a slave to the PDCS GUI interface. The slave nodes only issue slave-related commands, and in most cases, slave nodes responds to commands sent to them from the master node. Generally, master type commands allow configuration of a slave node or request status information from a slave. Slave messages generally consist of slave nodes reporting status information or streaming acquired data from their sensors. A slave node can also generate error-related messages if it detects a system error. Section 9.8 presents more detailed definitions of the master and slave commands. The primary purpose of the master PDSM board is to act as a conduit to move commands and data to and from the PDCS and the slaves.

The following pseudo code describes the general behavior of the master and slave nodes. Upon powering up, the *InitSystem()* function attempts to initialize all of the nodes peripheral and I/O devices. If there is an initialization failure, the system terminates execution and displays a pattern of blinking LEDs on the PDSM to indicate a failure. If power up is a success, the PDSM node lights the red LED to indicate success, and then goes into a sleep mode using the *sleepUntilMsg()* function and waits for a command to be received. The *sleepUntilMsg()* function returns when a new message is in the message buffer for processing. The *receiveMsg()* function is called to receive the message into a receive message buffer. It is the *receiveMsg()* function that handles all communications mediums, i.e., I2C, USB, or IEEE 802.15.4. Upon returning from *receiveMsg()*, the command is then processed with a command lookup table. In this case, the switch statement acts as the lookup table to process the incoming message. The incoming command is compared to those on the switch state, and when a match is found, the command is processed accordingly. In this pseudo code, the “do cmd” statements are place holders for the actual code that will be called.

We provide two examples in pseudo code to expose some detailed requirements of the communications. In the first example, the command **cmdGetStatus** is received and a *sendReplyMsg()* function is called with a value myStatus. The *sendReplyMsg()* function is designed to reply back to the originator of the command request with the requested information, which is the node’s status information in this case. The second example features an operating mode where a slave node, in particular, is commanded to take data by reception of the **cmdAcquireData** command as shown in the pseudo code. Upon receiving this command, the slave acquires data blocks as shown in the forever loop. After acquiring each data block, the node then checks for any pending messages by calling *checkForMessage()*, and if a message is pending, it stops acquiring data and services the pending message. This example shows the general processing flow of how the system is implemented in the PDSM prototype and may require some modification to get different behaviors. For example, a user may not want data acquisition to resume after a new message is processed. However this approach was not implemented in order to minimize design complexity.

What follows is the pseudo code for RMS initialization and the processing of incoming messages:

```
InitSystem(); //init all peripherals and code
for(;;)
{
    sleepUntilMsg();
    receiveMsg();
    switch(cmd)
    { //begin switch
      case: cmd1
         do cmd1;
          break;
      case: cmd2
         do cmd2;
          break;
      case: cmd3
         do cmd3;
          break;
         …
      case cmdConfigureSensors:
         configureSensors(configuration);
         break;
      case cmdGetStatus:
         sendReplyMsg(myStatus);
         break;
      case cmdAcquireData:
          forever
          {
              acquireDataBlock();
              if(checkForMessage()) break;
           }
      case: cmdN
         do cmdN;
          break;
     default:
         do invalidCmd
          break;
    }//end switch
}
```

9.7 Sensor Configuration

The sensor configuration message is an important message sent to the nodes that defines the context in which a node will operate when it receives an acquire data command. The configuration message defines several parameters such as the active sensors, sensor sampling rates, samples per data block on each sensor, sensor sampling interval, plot settings, and data archive settings. For more details on this implementation, see the *msglib.h* header file.

A key weakness to this approach is that as additional sensors are designed into the PDSM, this message format will have to change, thus significantly affecting software throughout the design. A better approach would be to define a configuration message for each individual sensor to

decouple sensors configurations from one another. The following is the top level data structure of a sensor configuration message:

```
typedef struct
{
  unsigned long SensorConfigMask;            //32 bit configuration mask ... see #defines below in msglib.h.
  unsigned long M3000SampleRateHz;           //sample rate
  unsigned long M3000NumSamples;             //samples per burst
  unsigned long OnBoardSampleRateHz;         //sample rate
  unsigned long OnBoardNumSamples;           //samples per burst
  unsigned long ThermCup1SampleRateHz;       //sample rate
  unsigned long ThermCup1NumSamples;         //samples per burst
  unsigned long ThermCup2SampleRateHz;       //sample rate
  unsigned long ThermCup2NumSamples;         //samples per burst
  unsigned long ThermCup3SampleRateHz;       //sample rate
  unsigned long ThermCup3NumSamples;         //samples per burst
  unsigned long VoltSampleRateHz;            //sample rate
  unsigned long VoltNumSamples;              //samples per burst
  unsigned long CurrentSampleRateHz;         //sample rate
  unsigned long CurrentNumSamples;           //samples per burst
  unsigned long AcquisitionIterval;          /not implemented
} SensorsConfigType;
```

The sensor configuration mask, SensorConfigMask in the SensorConfigType structure, is a 32-bit word. Each bit in the word represents some aspect of a sensors configuration as follows:

Bit 0 - enable M3000 x axis
Bit 1 - enable M3000 y axis
Bit 2 - enable M3000 z axis
Bit 3 - enable multiplex M3000 xyz accelerometer acquisitions
Bit 4 - archive all acquired M3000 data
Bit 5 - enable plot all acquired or playback data
Bit 6 - enable onboard x axis accelerometer
Bit 7 - enable onboard y axis accelerometer
Bit 8 - enable onboard z axis accelerometer
Bit 9 - multiplex Onboard xyz accelerometer acquisitions – not implemented
Bit 10 - archive onboard accelerometer data
Bit 11 - plot onboard accelerometer data
Bit 12 - sel1, onboard sensitivity level bit 0 control
Bit 13 - sel2, onboard sensitivity level bit 1 control
Bit 14 - enable thermocouple 1
Bit 15 - archive thermocouple 1 data
Bit 16 - plot thermocouple 1 data
Bit 17 - enable thermocouple 2
Bit 18 - archive thermocouple 2 data
Bit 19 - plot thermocouple 2 data
Bit 20 - enable thermocouple 3
Bit 21 - archive thermocouple 3 data
Bit 22 - plot thermocouple 3 data
Bit 23 - enable voltage test point
Bit 24 - archive voltage data
Bit 25 - plot voltage data
Bit 26 - enable current test point
Bit 27 - archive current data
Bit 28 - plot current data

Bit 29 - enable save data as ASCII, not implemented
Bit 30 - enable simple diagnostics, implemented, but very simplistic for demo purpose
Bit 31 - enable if time range selection is used during playback/retrieve data – works if real time clock functional

A design decision was made to minimize memory use, and so the 32-bit word SensorConfigMask was devised to control configuration of the sensors. Using the implementation as above, it soon became clear this size of this 32-bit mask becomes too limiting when the need to add more configuration related functionality to the sensors arises. In other words, the 32 bits soon become used up. Future implementation should look at this aspect more carefully and devise a better approach for sensor configuration. One approach could be to use more 32-bit words or an XML-based configuration dictionary. There could be many other approaches, but these comments are made for consideration in future designs.

9.8 Network Commands

For communications, all network commands and command definitions need to be defined. These are defined in the msglib.h file with the enumerate data structure called PdCommandSet and are relisted below. Most of these commands are not implemented, but are presented as a possible roadmap for potential types of commands that one might consider implementing in future development. For more details of how the commands are used, refer to the source files msglib.h and msglib.c. Note: There is a distinction between master and slave commands, in that commands that are intended to be issued by the master node begin with an “m” and commands intended to be issued by slave nodes begin with an “s”. This requirement must be enforced by the programmer. Generally, slave commands are sent in response to requests made by the master, or if a slave is reporting on its status.

```
enum PdCommandSet //2 byte command word
  {
   //master command set
   mCmdRequestStatus,                       //cmd
   mCmdStop,                                //cmd
   mCmdResetBoard,                          //cmd
   mCmdAcquire,                             //cmd
   mCmdAcquireWithRealTimeDataRequest,     //cmd
   mCmdConfigureSensors,                    //cmd, 32bit sensor config mask
   mCmdRetrieveAcquiredData,                //cmd, 32bit sensor config mask
   mCmdSetRTclock,                          //cmd, struct RTClockConfig
   mCmdSetLED1,                             //cmd
   mCmdSetLED2,                             //cmd
   mCmdSetLED3,                             //cmd
   mCmdReadSDMemBlock,                      //cmd, unsigned long blockNo
   mCmdReadSDMemHeader,                     //cmd
   mCmdClearSDMemBlock,                     //cmd, unsigned long blockNo
   mCmdCalibrateSensors,                    //cmd
   mCmdCalibrateSensor,                     //cmd, uchar sensorID
   mCmdReadRealTimeData,                    //cmd
   mCmdEnableExternalTrigger,               //cmd
   mCmdDisableExternalTrigger,              //cmd
   mCmdCycleLEDS,                           //cmd

   //slave command set
   sCmdVal,                                 //cmd, string
   sCmdTherm1Val,                           //cmd, short v
   sCmdTherm2Val,                           //cmd, short v
   sCmdTherm3Val,                           //cmd, short v
   sCmdThermAllVals,                        //cmd, short v1, short v2, short v3
```

```
  sCmdM3000AccelXYZvals,                //cmd, short v1, short v2, short v3
  sCmdOnBrdAccelXYZvals,                //cmd, short v1, short v2, short v3
  sCmdVoltageVal,                       //cmd, short v
  sCmdCurrentVal,                       //cmd, short v
  sCmdSDMemBlock,                       //cmd, struct memblock
  sCmdSDHeaderBlock,                    //cmd, struct headerblock
  sCmdRealTimeData,                     //cmd, struct realTimeData
  sCmdAck,                              //cmd, short cmdAck
  sCmdInvalid,                          //cmd, short, received invalid command
  sCmdAcquireNoSensorSelected,          //cmd, received an mCmdAcquire cmd, but no sensor on this slave is enabled.
  sCmdStatusReport,                     //cmd, null terminated data string reporting node status
  sCmdM3000XADCdata,                    //cmd, data ... subset of M3000 X axis sensor unscaled adc data to GUI
  sCmdM3000YADCdata,                    //cmd, data ... subset of M3000 Y axis sensor unscaled adc data to GUI
  sCmdM3000ZADCdata,                    //cmd, data ... subset of M3000 Z axis sensor unscaled adc data to GUI
  sCmdM3000XYZADCdata,                  //cmd, data ... subset of M3000 multiplxed xyz-axis sensor unscaled adc data to GUI
  sCmdOnBoardXADCdata,                  //cmd, data ... subset of OnBoard X axis sensor unscaled adc data to GUI
  sCmdOnBoardYADCdata,                  //cmd, data ... subset of OnBoard Y axis sensor unscaled adc data to GUI
  sCmdOnBoardZADCdata,                  //cmd, data ... subset of OnBoard Z axis sensor unscaled adc data to GUI
  sCmdOnBoardXYZADCdata,                //cmd, data ... subset of OnBoard multiplxed xyz-axis sensor unscaled adc data to GUI
  sCmdVoltsADCdata,                     //cmd, data ... unscaled adc voltage data to GUI
  sCmdCurrentADCdata,                   //cmd, data ... unscaled adc current data to GUI
  sCmdThermalCouple1,                   //cmd, data ... thermalcouple 1, deg C measurement, 4byte long
  sCmdThermalCouple2,                   //cmd, data ... thermalcouple 2, deg C measurement, 4byte long
  sCmdThermalCouple3,                   //cmd, data ... thermalcouple 3, deg C measurement, 4byte long
  sCmdThermister,                       //cmd, data ... thermalcouple 1, deg C measurement, 4byte long
  sCmdStatusMemoryCardFull,             //cmd --report that memory card is full
  sCmdStatusMemoryCardByteSize,         //cmd, unsigend long - byte size ... implies support of up to 4GB ????
  sCmdStatusMemoryCardAvailableBytes,   //cmd, unsigend long - byte size ... implies support of up to 4GB ????
  SMemoryCardNotDetected,               //cmd
  sMemoryCardInvalidSize,               //cmd
  sMemoryCardInvalidFATFormat,          //cmd
  sMemoryCardInvalidDirectory,          //cmd
  sCmdOnBoardXYADCmean,                 //cmd, int16-x, int16-y - unscaled average readings across x&y axis of onboard acceler
//new commands ... slave fault detection commands
  sCmdTC1TemperatureFault,              //Cmd, tval, tmin,tmax - slave reports temperature out of bounds of limits
  sCmdTC1TemperatureMinFault,           //Cmd, float tval, float tmin   - slave reports temperature below lower limit, units deg F
  sCmdTC1TemperatureMaxFault,           //Cmd, float tval, float tmax - slave reports temperature above max limit, units deg F

  sCmdTC2TemperatureFault,              //Cmd, tval, tmin,tmax - slave reports temperature out of bounds of limits
  sCmdTC2TemperatureMinFault,           //Cmd, float tval, float tmin   - slave reports temperature below lower limit, units deg F
  sCmdTC2TemperatureMaxFault,           //Cmd, float tval, float tmax  - slave reports temperature above max limit , units deg F

  sCmdTC3TemperatureFault,              //Cmd, tval, tmin,tmax - slave reports temperature out of bounds of limits
  sCmdTC3TemperatureMinFault,           //Cmd, float tval, float tmin  - slave reports temperature below lower limit, units deg F
  sCmdTC3TemperatureMaxFault,           //Cmd, float tval, float tmax  -slave reports temperature above max limit , units deg F

  sCmdVoltageFault,                     //Cmd, vval, vmin,vmax - slave reports voltage out of bounds of limits
  sCmdVoltageMinFault,                  //Cmd, vval, vmin      - slave reports voltage fell below lower limit
  sCmdVoltageMaxFault,                  //Cmd, vval, vmax      - slave reports voltage rose above upper limit

  sCmdCurrentFault,                     //Cmd, float cval, float cmin, float cmax - slave reports current out of bounds limits
  sCmdCurrentMinFault,                  //Cmd, float vval, float vmin      - slave reports current fell below lower limit
  sCmdCurrentMaxFault,                  //Cmd,  float vval, float vmax      - slave reports current rose above upper limit

// error commands
  sCmdVoltageFileDataHeaderError,       //Cmd - data block header error
  sCmdCurrentFileDataHeaderError,       //Cmd - data block header error
  sCmdTemperatureFileDataHeaderError,   //Cmd - data block header error
  sCmdExternAccelFileDataHeaderError,   //Cmd - data block header error
  sCmdOnbrdAccelFileDataHeaderError,    //cmd - data block header error
  sCmdFileDataReadError,                //cmd - generic error reading data file

//more new commands – command the slave uses for sending acquired data back to the GUI.
  sCmdArchivedM3000ADCdata,             //cmd header, data block header, data
  sCmdArchivedM3000XADCdata,            //cmd header, data block header, data
  sCmdArchivedM3000YADCdata,            //cmd header, data block header, data
  sCmdArchivedM3000ZADCdata,            //cmd header, data block header, data
  sCmdArchivedM3000XYZADCdata,          //cmd header, data block header, data
  sCmdArchivedOnBoardADCdata,           //cmd header, data block header, data
  sCmdArchivedOnBoardXADCdata,          //cmd header, data block header, data
```

```
  sCmdArchivedOnBoardYADCdata,          //cmd header, data block header, data
  sCmdArchivedOnBoardZADCdata,          //cmd header, data block header, data
  sCmdArchivedOnBoardXYZADCdata,        //cmd header, data block header, data
  sCmdArchivedVoltsADCdata,             //cmd header, data block header, data
  sCmdArchivedCurrentADCdata,           //cmd header, data block header, data
  sCmdArchivedThermalCouple1,           //cmd header, data block header, data
  sCmdArchivedThermalCouple2,           //cmd header, data block header, data
  sCmdArchivedThermalCouple3,           //cmd header, data block header, data
  sCmdArchivedThermalCouple,            //cmd header, data block header, data
  sCmdPlotData,                         //cmd header, data block header, data ... plot it.
 };
```

9.9 Wireless Communication Firmware Description

9.9.1 Digital Communication via a Serial Peripheral Interface

The digital interface between the MCU and transceiver allows the MCU to configure the transceiver into different modes, read and write buffered data, and read back transceiver status information. This communication is provided by SPI. Figure 28, taken from the CC2420 datasheet, illustrates the SPI bus interface between the CC2420 transceiver and MCU. The CSn, SI, SO, and SCLK pins comprise the 4-pin SPI bus while the FIFO, FIFP, CCA, and SFD pins allow the software to monitor the status of the TXFIFO and RXFIFO as well as the start of frame delimiter and clear channel assessment pins.

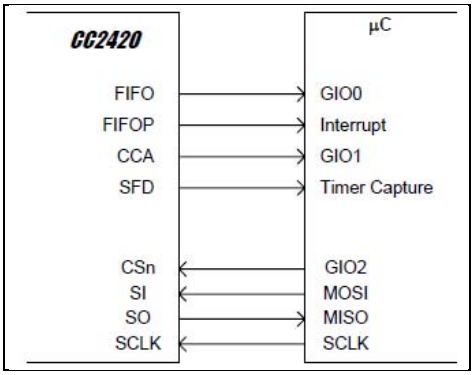


Figure 28. SPI interface between the transceiver and MCU (*1*).

For more details on the 4-pin SPI interface see the Chipcon CC2420 datasheet (*1*) and the cCC2420 source code.

9.9.2 cCC2420 Class Structure Descriptions

The structures within the CC2420 class define the attributes of the data packets for transmission, reception, and the network in general. BASIC_RF_TX_INFO defines the data structure, which is used to transmit packets as follows:

```
typedef struct {
   WORD destPanId;       // network PAN ID
   WORD destAddr;        // address of intended receive node
   INT8 length;         // length of transmitted packet payload
   BYTE *pPayload;       // transmit packet payload
```

```
   BOOL ackRequest;      // wireless acknowledgement enable
   BYTE haa;             // appended payload header, byte 1
   BYTE hff;             // appended payload header, byte 2
   UINT16 dataLength;    // length of entire appended payload
   WORD storageIndex;    // position within appended payload
} BASIC_RF_TX_INFO;
```

BASIC_RF_RX_INFO defines the data structure, which is used to receive packets as follows:

```
typedef struct {
   INT8 length;          // length of received packet payload
   BYTE seqNumber;       // order of received packets
   WORD srcAddr;         // address of node that sent packet
   WORD srcPanId;        // network ID
   WORD destAddr;        // address of intended receive node
   BOOL ackRequest;      // wireless acknowledgement enable
   INT8 rssi;            // received signal strength
   BYTE *pPayload;       // received packet payload
   //BYTE *pMsgData;       // appended message payload
   //BYTE haa;             // appended payload header, byte 1
   //BYTE hff;             // appended payload header, byte 2
   //UINT16 dataLength;    // length of entire appended payload
   //WORD storageIndex;    // position within appended payload
} BASIC_RF_RX_INFO;
```

BASIC_RF_SETTINGS defines the settings used generally by all nodes in performing both wireless transmissions and receptions:

```
typedef struct {
   BASIC_RF_RX_INFO *pRxInfo;    // receiption struct (see above)
   UINT8 txSeqNumber;            // order of transmitted packets
   volatile BOOL ackReceived;    // indicates whether a wireless acknowledgment is received
   WORD panId;                   // network ID
   WORD myAddr;                  // node address of self
   BOOL receiveOn;               // indicates whether CC2420 is in receive mode
   BYTE  messageReady;           // goes high to indicate a new message is ready
} BASIC_RF_SETTINGS;
```

The cCC2420 software class was defined to include all relevant functions and structures that pertained to the operation of the CC2420 transceiver chip. For more information on the functions relating to the operation of the CC2420 transceiver, see the cCC2420.cpp source file.

9.10 SD Card Data Storage

This section documents the general data storage format on the SD memory card. The biggest storage sizes of memory cards used with the PDSM prototype was 2 GB; however, larger cards can be used. For easy PC access to the data stored on the card, we decided to use a FAT32 file system on the memory card. Although this has major advantages, a key disadvantage of FAT32 is that the I/O speeds are not as fast as using raw file I/O. If I/O storage rates are too slow when using a FAT32 device driver, it may be possible to tweak the FAT32 device driver where

appropriate to achieve faster access times. For purposes of the prototype demonstration, I/O storage rates were not a prime consideration.

Because of some limitations of the FAT32 driver used, empty directories for each sensor type were created on the SD memory card with a PC workstation before inserting the SD card into the PDSM memory slot. The SD card directory names were created on the PC using a simple bat script file running the following commands on the memory card:

```
mkdir ONBRDVIB
mkdir XTRNLVIB
mkdir TEMPER
mkdir VOLTAGE
mkdir CURRENT
```

The PDSM firmware expects these directories to exist before it can properly store data to the SD cards. In a complete FAT32 firmware library, creating directories on the PDSM should be possible. When a PDSM board is commanded to archive data, it creates the following files for each sensor type if they do not already exist:

```
ONBRDVIB/data.bin", //contains onboard vibration data measurements
XTRNLVIB/data.bin", //contains external M3000 Vibrametrics measurements
TEMPER/data.bin", //contains thermocouple measurements
VOLTAGE/data.bin", //contains voltage test point measurements
CURRENT/data.bin", //contains current test point measurements
```

If the data file already exists when a PDSM is commanded to store a data set, the data are automatically appended to the file. This is done to preserve previous acquisitions. Which sensor data is stored during acquisitions depends on how the PDSM board has been configured from a command sent by the PDCS GUI. Presently there is no command implemented to delete files from the card. It is conceivable that this would be a useful feature in future development, but it was not done in this implementation because of the FAT32 device driver limitations.

Each data file has a well-defined data storage format. Each sample set of data for each sensor is written to the file as a block of data. The data are stored as sequential sets of data blocks, which consist of the data block header, followed by the raw sensor data. The data storage structure of the file is as follows, where BlockM is the maximum number of data blocks in the file:

```
Block1
        DataBlockHeader;
        DataBlock
 Block2
        DataBlockHeader;
        DataBlock
Block3
        DataBlockHeader;
        DataBlock
       …
BlockM
        DataBlockHeader;
        DataBlock
```

The DataBlock is the actual data acquired from the configured sensor, and its context is defined by the DataBlockHeader. The DataBlockHeader is defined as follows:

DataBlockHeader

SyncPattern_aa_55h – 2 byte syncronization pattern for data integrity.
BlockLength – 2 bytes – length in bytes of DataBlockHeader & data - of allows up to 65k byte block length, although this can be restricted by processor RAM limitations.
SampleRateHz – 4 byte uint, sample rate in Hz of the data .
Sensor – 1 byte – identifies the sensor data was acquired from (one of eTC1, eTC2,eTC3, eVoltageTp, eCurrentTp, eExternalAccelX, eExternalAccelY, eExternalAccelZ, eExternalAccelXYZ, eOnboardAccelX, eOnboardAccelY, eOnboardAccelZ, eOnboardAccelXYZ)
SampleUnits - 1 byte sensor measurement units: once the data values are multiplied by SampleScaleFactor the data will be in units of SampleUnits. This will be an enumerate type of type eVoltsUnits, eAmpsUnits, eGsUnits, eCelciusUnits, eFarenheightUnits
SampleScaleFactor – float type – 4 bytes scale factor for the acquired data. Definition is "Sensor" specific. Multiplier to convert raw data to units of volts, current, degC, G's, etc.
NumSamples – 2 bytes number of total samples in this block of data.
EpochTimeStamp – 4 byte time stamp –epoch time is seconds since Jan 1, 1970 when data was acquired.

The following is a key point. The design approach has focused primarily on the flexibility of storage, not on storage speed or efficiency. There are obvious cases where there is significant data header overhead. As an example, when measurements are taken on a thermocouple, single point measurements are typically taken over periods of seconds, minutes, or over greater time periods due to the nature of slow temperature changes. In the case of the PDSM design, this is due to the slow sample rate of the ADCs attached to the thermocouples. For every 2-byte thermocouple measurement taken, there is an overhead of 20 bytes for the data block header, which amounts to 90% of the data block. In a second example, where the header is not significant, if samples are taken on the current sensor, one might take 512 2-bytes samples per block. This would lead to a header overhead of ~2% of the data block, which is more attractive. The point to these remarks is to make the developer aware of the overhead tradeoffs and prompt the developer to be open to exploring some other approach to a data storage format that may offer better storage efficiency.

Another point of interest relates to the required accuracy of the timestamp applied to the data block header. The timestamp represents the time at which a data block's acquisition began. For this prototype, 1 s was a reasonable resolution. However, one must be certain what an acceptable resolution is for a particular application. Knowing this in advance will drive requirements on the systems hardware design.

There are some additional enhancements on the file format that need to be considered. Very likely, there should be a file header that provides some additional information that centers around the notions of metadata such as an ASCII text block describing the nature of what is being

measured, a parameter identifying the board address that the data was acquired from to address the possibility of moving memory cards from one PDSM board to another, among others. These concepts should be designed into the next revision.

10. User's Manual

Here we provide an overall description of the user operation of the PDSMs. This section covers the PDSM's I/O capabilities, sensors types, sampling rates, configuration options, data storage formats, LED meanings, reset button use, power up state, GUI interface description, board power up, how the board is intended to be operated, intercommunications, and playback operations.

10.1 Hardware Manual

10.1.1 PDSM Board Jumpers for I2C Communications

When using I2C communications, the master and slave jumpers must be set correctly, as shown in figure 21. On the master board only, on the P12 connector, one must jumper the SDA and SCL pins correctly by setting a jumper across pins 1 and 2. This is shown with the orange arrow in figure 29. This jumper attaches a 3.3-V pull up resistor to the SDA line. For the SCL clock, one must set a jumper across pins 3 and 4 to attach a 3.3-V pull-up resistor to the SCL clock line. For all boards performing I2C communications, the user must connect a daisy-chained ribbon cable from board to board using the P8 connector on each bard. This setup is shown with the blue arrows in figure 29, where the yellow, black, and red cable interconnect two boards by connecting the SDA, SCL, and GND pins of each board together. On must be certain to interconnect the ground pins to minimize noise and establish a common ground between the boards. Do not place any jumpers on the p12 connectors on the slave boards. This is indicated on the slave board in the left of figure 29.

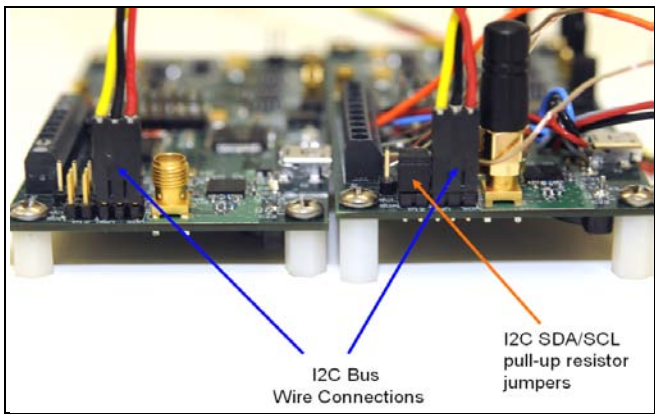


Figure 29. I2C connections between two boards.

10.1.2 Thermocouple Sensors

Figure 30 shows the thermocouples connecting to the screw terminals. The PDSM supports the connection of three K-type thermocouples. The thermocouples used in this design were the Omega part number SA1-K-72. Although these thermocouples are rated to handle temperatures in the range from –75 to 350 °F, the board's firmware is only designed to handle temperature readings from –8 to 334 °F. These limits have not been tested and must be confirmed if an application requires such a wide range of temperature measurements. Also, the accuracy of the temperature measurements is about ±1 °F of error as noted in laboratory measurements. The three thermocouples are attached to the PDSM board's terminal connector P6, using pins 1 through 6. Thermocouple 1 connects to pins 1 and 2, thermocouple 2 connects to pins 3 and 4, and thermocouple 3 connects to pins 5 and 6. Because of the nature of thermocouple polarity, be sure the terminals are connected correctly for proper measurements. One key point is, what are the actual temperature limits that would be required for a given application? This choice could affect the temperature conversion routines as well as the actual thermocouples used in the system. Figure 30 shows the three pairs of thermocouples wires connected the screw terminal inputs for thermocouples one, two, and three (denoted with arrows TC1, TC2, and TC3).

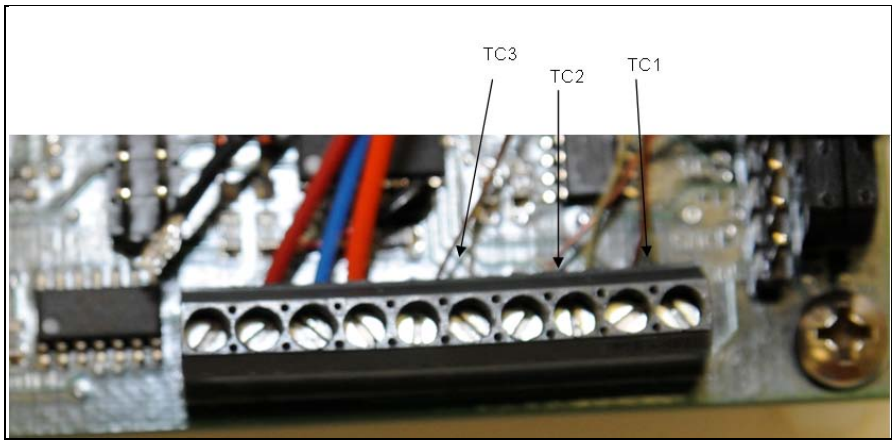


Figure 30. Thermocouple wires connected to screw terminal.

10.1.3 Reset Button

Onboard the PDSM is a reset button. This button is provided in the event of the PDSM board locking up, which occurs relatively frequently in the present design. Pressing the reset button causes the MSP430 to cycle on a boot-up sequence. In the present release of the firmware, upon a reset or power up, the PDSM comes up in a wait state where it is waiting to respond and process commands received on one of its communications mediums. A board that successfully boots up displays a lit solid red LED.

10.1.4 LED Status Lights

The system uses LEDs to give the user a visual on the real-time status of the PDSM. On the present design, there are three LEDs: red, blue, and yellow. Each LED is used for a specific purpose, and table 7 gives definitions of their functionality.

Table 7. Overview of the red LEDs status blinks.

Red LED Blink Counts	Meaning
1	Memory Card Not Detected
2	Memory Card Initialization Failed
3	Memory Card Invalid Size
4	Memory Card Invalid FAT Format
5	Memory Card Invalid Directory
6	Memory Card Invalid File
7	Memory Card Invalid File or Directory
8	Memory Card Write Failure
9	Others as required

10.1.5 Red LED

A flashing red LED indicates a fatal system failure mode. If this LED is blinking, the P&D board is in a fatal system failure state. This state cannot be resolved without the user taking some physical action on the system. Examples of such a failure would be that the memory card is not inserted or the memory card is not properly formatted. When blinking, the LED blinks a certain number of times within a certain time interval. The number of blinks indicates the failure mode as outlined in table 7. When in a fatal system failure mode, all other system functions are disabled, and the user must resolve the error and reset the board.

10.1.6 Yellow LED

A lit yellow LED means that the PDSM board is acquiring and storing data to the SD memory card. When the LED is not lit, acquisition is not being performed.

10.1.7 Blue LED

A lit blue LED is lit indicates that the PDSM board is performing communications on I2C, USB, or the IEEE 802.15.4 interface. When the LED is not lit, these functions are not being performed.

Although we used LEDs to indicate the PDSM board's status, future designs should consider using other indicators on the PDSM that may be more user friendly. One possibility is using a low-power liquid crystal display (LCD) to show ACSII status messages and report on sensor measurements without needing to feed that data back to the GUI. This area should be further investigated in future designs.

10.2 GUI Manual

The prototype GUI, shown in figure 31, represents the functionality of a PDSM. The goal with this GUI design was to provide a simple but flexible user interface to give complete command and control over all PDSM boards in the network. Although the interface was developed in C/C++, it can be developed in any language as long as it implements the proper communications messages as defined in this documentation.

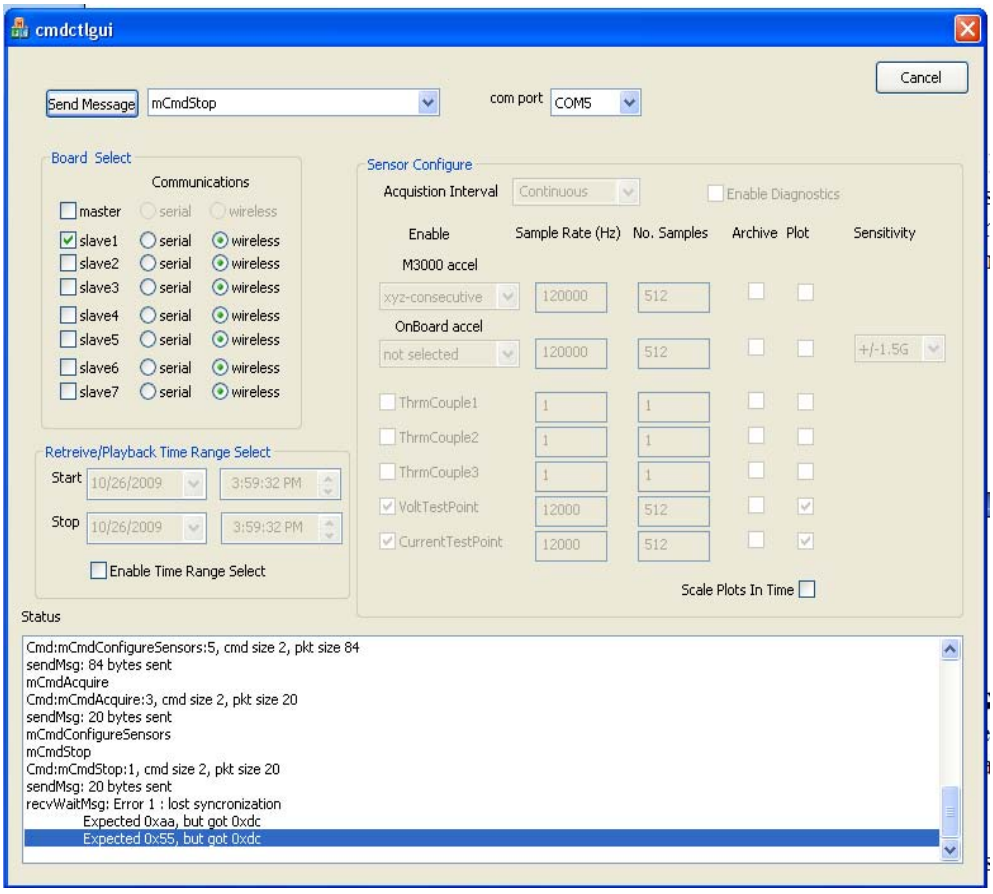


Figure 31. GUI used to configure the prototype PDHMS PDSM network.

10.2.1 Communication Port Selection and Master Node Configuration

As shown in figure 31, the PDSM can be remotely configured and controlled. The interface presently supports up to eight nodes. The master node must always have a node address of 0. This is done by removing jumpers JMP0, JMP1, and JMP2. The master node must always be attached to the PDCS running the GUI via a USB cable. Before communicating from the PDCS to the master node, the "Com Port" selection must be made. This allows the user to select which virtual communication port the master PDSM node is attached to on the PDCS. The user should issue a **cycle LEDs** command to the master node as a simple test to verify that the master has been successfully connected. The LEDs should cycle until the user issues a **stop** command to the master node. If this does not work, check the selected virtual communication port and power

levels on the master node. Once successfully connected, the master node can then communicate to the desired slave nodes either wirelessly or via I2C. Although the master node can be configured to acquire data, we recommend not doing this, primarily because the system can get overloaded and tends to drop communications messages when acquiring data and simultaneously handling communications between the PDCS GUI and/or slave PDSMs.

10.2.2 Slave Node Selection

Each PDSM slave board can be configured to operate wirelessly or serially. A board is configured as a slave by setting the board ID jumpers to value 1 through 7. If a board is to communicate to the master via I2C, then the master and the slave boards must be wired together via the I2C serial bus. For wireless communications, the boards' antennas must be connected to the SMA connectors.

The "Board Select" group buttons allow the operator to select which PDSM boards the GUI commands will be sent. The operator can do this by checking or un-checking the master or one or more of the slave checkboxes. Once done, the operator can select which medium to use for communicating with the desired PDSM board (either wireless or serial). Selecting wireless means the workstation communicates wirelessly from the master node to the associated PDSM slave board. Selecting serial means the PDCS communicates from the master to the selected slave via I2C. For wireless communication to work, an antenna must be attached to the PDSM board. For serial communication to work, the slave PDSM board must be wired to the master PDSM board via the I2C connections on the boards. A group of slaves can be configured at once, or the operator can selectively pick which slaves with which to communicate.

The present design does not allow simultaneous communication between both wireless and I2C mediums. Based on the radio box selections, the master node automatically communicates to the desired node on the selected communication medium. To confirm successful communication to a slave PDSM, the operator can issue a cycle LED command to the desired slave or make a status request to receive the slave PDSM's status information. If the slave does not respond to these requests, one should check the board power levels, slave jumper settings, and I2C connection, or confirm that the nodes have their antennas connected.

Through the GUI, the operator can enable or disable the desired sensor, select the sensors' sampling rates, select whether or not to archive the acquired data, and select whether or not to plot the data with the MATLAB display.

10.2.3 Sending Messages to the Nodes

The Send Message button and the drop-down selection box allow the operator to select the desired command. Once selected, the user clicks on the Send Message button. The selected message is sent to the selected PDSM board. The dropdown list commands include **mConfigureSensors**, **mCmdStop**, **mCmdAcquire**, and **mCmdPlayback**, which are used to configure, stop, start, and playback selected data in the PDHMS network, respectively. There are

other commands available and other commands can be easily added to the interface as required. In the present design, when a new command is issued to a board, the board will stop whatever it is doing, and then process the received command. For example, if a board is acquiring data and receives a status request command, it will stop acquiring data and process the status request command. After processing the status request command, it will then remain in the stop state. It will not resume its previous acquisition until a new acquire command is issued to the node. Future implementations will likely have the nodes resume an acquisition after processing a new incoming command. The actual behavior will depend on the context of the new command and the context of its current state. This aspect will require careful design.

10.2.4 Receiving Messages from the Nodes

As nodes perform their tasks, nodes may generate messages of one type or another. For instance, a node may automatically generate error messages to the GUI if, for example, the mode attempts to write data to the SD memory card and fails. Messages from the nodes to the GUI can be composed of packets of acquired data taken from the PDSM's sensors as well as simple real-time diagnostics messages.

10.2.5 Simple Diagnostics

Selecting the Enable Diagnostics button activates the ability to check collected sensor data against some threshold crossing in real time. If values exceed the predefined threshold settings, then error messages will be communicated back to the GUI. The diagnostics messages are based on very limited diagnostics routines that do threshold detections. This implementation was designed for proof of concept demonstrations and requires further development of more sophisticated algorithms and GUI controlled configuration parameters. The primary idea behind this limited implementation was to demonstrate that real-time diagnostics algorithms could be configured to scan for fault conditions in the platform and report on these faults.

10.2.6 Configuring the Sensors of Each Board to Acquire Data

The Sensor Configure group is used to configure the sensors on each of the boards that are selected in the Board Select group. The operator can turn a sensor on or off by checking the box in the "Enable" column. The operator can also set the desired sample rate and the number of samples per block of samples taken, as well as the Archive or Plot buttons to archive or display the data as it is acquired or during playback. The Sensitivity drop-down box is used to configure the sensitivity of the onboard accelerometer.

The "Acquisition Interval" is a key configuration parameter. It defines the periodicity at which the sample blocks are taken. The period can be seconds, minutes, hours, days, months, or years. This board will wake up at these intervals, take the configured measurements, and then go back to sleep. This feature was not implemented in this release.

10.2.7 Status Window

The Status window shows the results of all requests made to the nodes and some of the information sent back from a node to the GUI. This window may contain the system status, sensor measurement results, warnings, or error messages. This status window effectively gives the user real-time feedback on the state of the desired PDSM boards. Further development should look more carefully at how information is presented to the user so that it is presented in the most meaningful way. Status messages should be standardized so that they are always presented in a consistent manner to the user to prevent confusion.

10.2.8 Data Retrieval and Playback

The PDSM design has a remote "Data Retrieval and Playback" functionality. This function gives the user remote access the data on the PDSM SD memory card through a communications medium, either IEEE 802.15.4 or I2C. This function eliminates the need for the user to have physical access to the SD card to view the data. Because IEEE 802.15.4 and I2C are relatively slow (at about 250 kbps), it does take considerable time to retrieve large data sets.

The Retrieve/Playback Range Time Select group on the GUI allows the operator to select the timeframe when retrieving acquired data from the PDSM boards. The data can be retrieved to the PDCS and plotted or stored locally for a more detailed analysis. The operator selects the start and stop playback times and then chooses a data retrieve command from the Command drop-down list. The data is then retrieved from the SD/MMC card of the selected PDSM board. Note: When retrieving data for playback or local storage, only select one node at a time because selecting more than one node and requesting data will overwhelm the network and cause many dropped packets. The network and the GUI interface currently implemented are not robust enough in handling the data load coming from multiple sources. This limitation needs to be addressed in future implementations. A possible solution would be to implement the full ZigBee stack for wireless communications and implement more robust communication protocols on the I2C bus interface.

When the GUI receives remotely requested data from the PDSM boards, it either saves the retrieved data to a local file or sends the data to the appropriate display for plotting. The GUI can be configured to save the data locally into separate files for each sensor. The files are automatically named to indicate from which PDSM board the data was retrieved. The remote PDSM sensor in playback or remote retrieval terminates its activity when commanded to do so by the remote GUI, even if it has not completed the previous retrieval request.

10.2.9 Storing Retrieved Data to PDCS

The primary goal of data storage on the PDCS is to accommodate data sets from multiple boards and store them in separate files. To simplify the storage process, the user is only able to select the file location, not the filenames. The retrieved data is stored in the format as described in section 9.10. Local file data storage is named using the following convention:

Filenames of Retrieved Data

Local directory\BRD# ONBRDVIB.data.bin
Local directory\BRD#XTRNLVIB.data.bin
Local directory\BRD#TEMPER.data.bin
Local directory\BRD#VOLTAGE.data.bin
Local directory\BRD#CURRENT.data.bin

where the # is replaced by the PDSM board's address ID. IDs in current design range from 0 through 7 and are implemented across three jumpers on each PDSM board.

10.2.10 MATLAB Displays

The MATLAB engine API was used to integrate the C/C++ programmed GUI with display routines and some simple MATLAB post-processing functions. Figure 32 shows the displays used for viewing the sensor data. MATLAB was integrated with the GUI interface to analyze the data in the time or frequency domain, which lays the foundation for performing post analysis and developing prognostics algorithms. The "Scale Plot In Time" option button is used for data display; one can switch between displaying the raw data with a time scale or displaying just the number of data samples per block. The displays implemented to date are primarily Cartesian displays for viewing the raw data and their frequency spectrum. Further investigation is needed to determine the type of displays from which a user may actually benefit most.

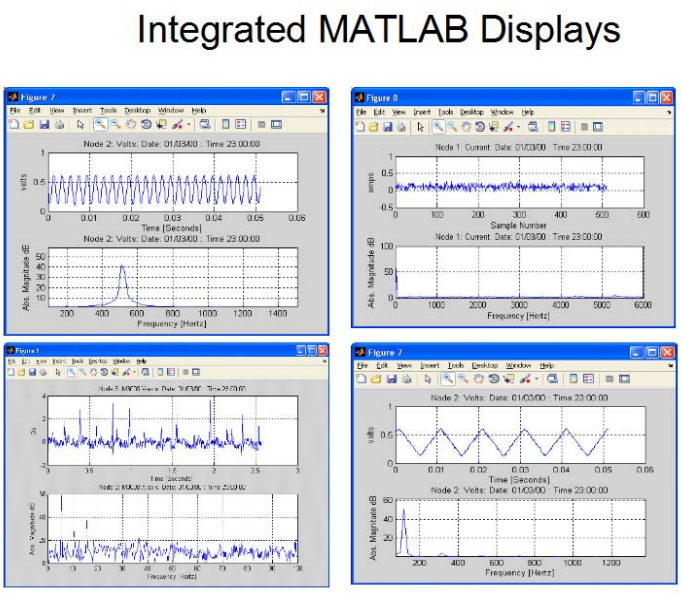


Figure 32. Real-time data displays.

10.2.11 Exiting the GUI

Clicking on the Cancel button exits the GUI application. Exiting the GUI does not affect the state of the master and slave nodes. The nodes remain in the state they were last commanded to be in prior to exiting.

---

11. General Performance Measurements

11.1 Vibration Experimental Results

11.1.1 Fault Simulator and Test Setup

To verify that the PDSM's data collection capabilities were functioning properly, we collected vibration data from a Machinery Fault Simulator from Spectra Quest, as shown in figure 33. This simulator provided a platform to generate vibration signatures for mechanical bearings of different sizes rotating at different frequencies, and in the case of these measurements, the gears were rotated at 20, 30, and 45 Hz. Data were collected using the PDSM and stored on the memory card. The data on the memory card was analyzed and compared to data collected using an eDAQ Lite Laboratory data acquisition system made by Somat, Inc. Both the PDSM and eDAQ Lite measured the data using a Vibra-Metrics Model 3000 miniature tri-axial accelerometer capable of sensing ±500 G's.

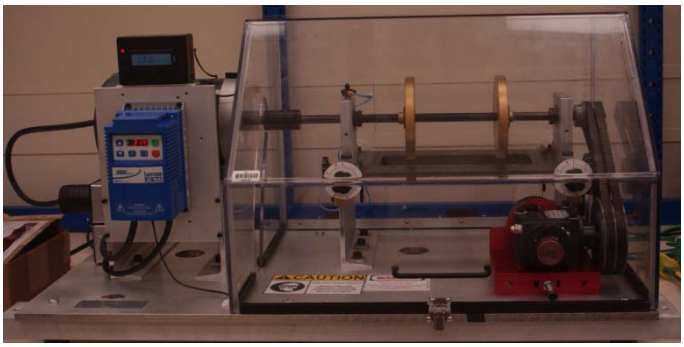


Figure 33. Machinery fault simulator used to determine bearings vibration signatures.

11.1.2 PDSM Data Acquisition Test Results

Frequency responses of the data for both systems were compared using an averaged Fourier Transform of the raw data. The data were normalized using the root mean squared (RMS) value and the DC bias was subtracted out before applying the Fourier Transform. Normalizing the raw data by the RMS value suppressed the noise within the signal, thus minimizing any contribution such noise would have on the vibration signature.

Due to data block size limitations on the PDSM board, the PDSM was limited to recording multiple 512 sample blocks of non-continuous data, whereas the eDAQ Lite system was able to stream continuous data without the 512 sample size limitation. To account for this discrepancy in the systems, individual Fourier Transforms were applied to 80 randomly selected data blocks, each containing 512 data points. The magnitudes of the Fourier Transform for each block of data were added and then averaged to produce the frequency responses.

These frequency responses represented the vibration signatures for one of the three axes for a healthy bearing. Figure 34a and b show the raw data collected by the PDSM board and eDAQ Lite data acquisition system, respectively. This data correspond to data collected on the *y*-axis of the tri-axial accelerometer. Data collection was performed for both systems in two separate runs on the fault simulator with identical setups at a sampling rate of 50 kHz. The frequency of rotation for the bearings was 45 Hz. The data were displayed in multiples of the gravitational constant in units of m/s^2. Differences in the magnitude can possibly be attributed to different noise levels in the two systems or to gain errors in the ADC data acquisition circuitry. However, differences in the magnitude of the raw data did not affect the frequency components of the signal.

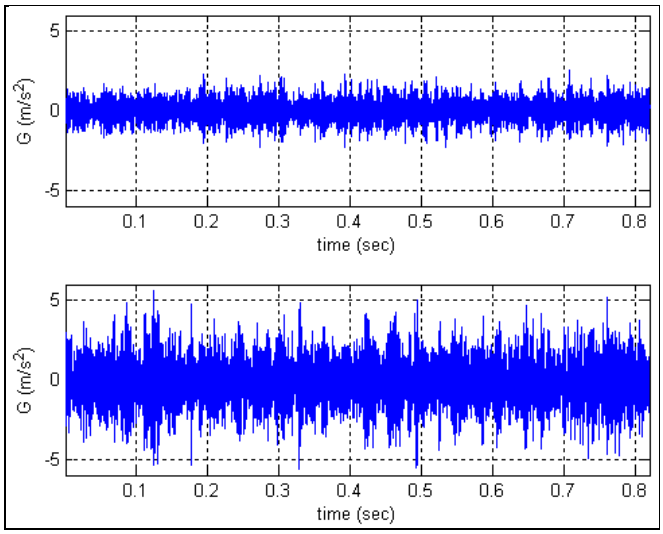


Figure 34. (a) Raw data for the y-axis collected by the PDSM application and (b) raw data for the y-axis collected by the eDAQ Lite.

Figure 35 shows the vibration signatures computed from the data shown in figure 34a and b. Because data were collected with the two different systems for two separate runs with the same setup parameters, some differences in the measured results were expected.

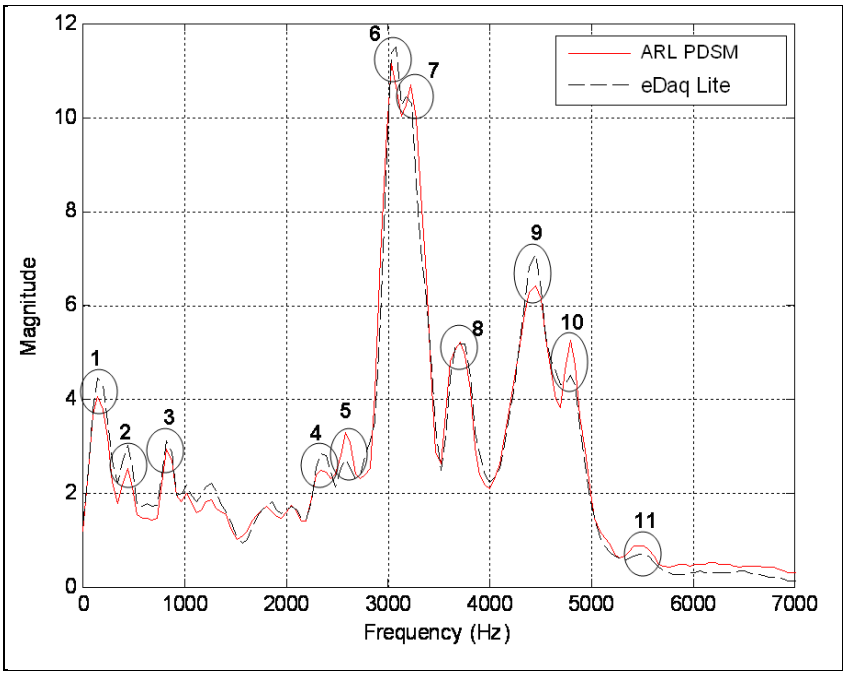


Figure 35. Overlaid vibration signatures for PDSM and eDAQ Lite data acquisition systems with the peaks of interest highlighted.

Table 8 shows that the numerical comparisons for peaks 1 through 11 of the two data acquisition systems exhibit a close correlation in frequency. The average magnitude differences between the PDSM and eDAQ Lite data are less than 10%. The magnitudes of the peaks are less important for the vibration signature than the accuracy of the peak frequencies. This data also show that the two vibration signatures will converge as they are integrated over an increasing number of data blocks.

Table 8. Vibration signature data comparison for PDSM and EDAQ lite data acquisition systems.

Peak	Freq 1 (Hz)	Freq 2 (Hz)	Difference (Hz)	Mag1	Mag 2	Difference
1	146.8	146.8	0	4.44	4.05	0.39
2	440.3	440.3	0	3.01	2.52	0.49
3	831.7	831.7	0	3.11	2.93	0.17
4	2348.0	2348.0	0	2.83	2.48	0.35
5	2593.0	2593.0	0	2.70	3.27	0.56
6	3033.0	3033.0	0	11.38	11.15	0.23
7	3229.0	3229.0	0	10.30	10.70	0.40
8	3718.0	3718.0	0	5.17	5.20	0.04
9	4452.0	4452.0	0	7.09	6.39	0.70
10	4795.0	4795.0	0	4.51	5.24	0.73
11	5528.0	5528.0	0	0.70	0.85	0.15

12. CROWS Demonstration

This section provides a very brief overview of the ARL PDHMS CROWS demonstration. The primary purpose of the exercise was to demonstrate real-time data collection and wireless transmission by the PDSM boards, testing operation, particularly when inducing a failure. As a final proof of concept demonstration, the ARL PDSM was installed into a CROWS at Picatinny Arsenal on October 30, 2009. In this proof of principle demonstration two PDSM slave boards were integrated into separate cavities of the CROWS, with a third master PDSM connected to the PDCS. The idea was to remotely control the PDSM boards using the wireless communications in the design. Once integrated into key locations of interest in the CROWS, the PDSM boards were remotely configured to monitor the control circuit cards in the CROWS using the PDCS command and control GUI. The PDSM boards monitored the accelerometer, voltage, temperature, and current data from each of the test points within the CROWS while the CROWS remained operational.

Figure 36 shows a PDSM module wired into the CROWS elevation control cavity. The jumper settings on the PDSM were set to node ID 2 and the PDSM was wired to the 28-V power source in the elevation cavity. The PDSM was wired to monitor the temperature of a resettable fuse using thermocouple 1, the temperature of the L-chip using the thermocouple 2, the main power voltage level using the voltage test point sensor, and the main power current using the current test point sensor. The cavity was left open for demonstration purposes, but could have been closed and sealed.

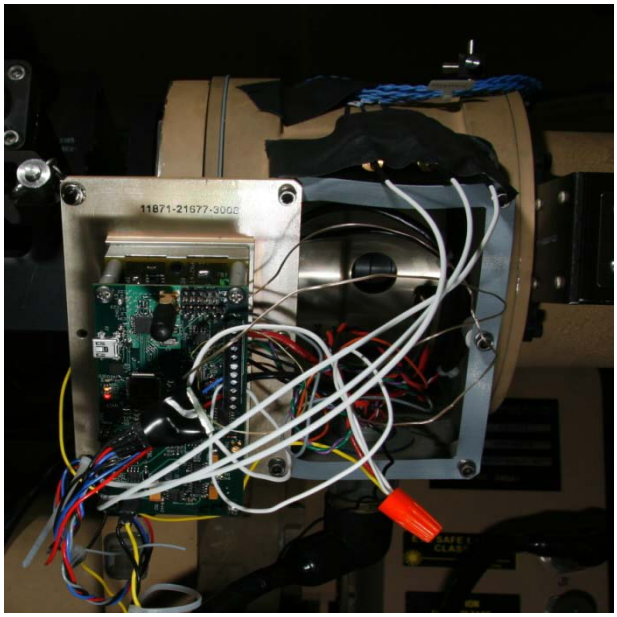


Figure 36. PDSM installed on the elevation control circuit card.

Figure 37 shows a PDSM module wired into the sealed CROWS SU motor/actuator cavity. The jumper settings on the PDSM were set to node ID 1 and the PDSM was wired 28-V power source in the cavity. The PDSM was wired to monitor the temperature of a resettable fuse using thermocouple 1, the temperature of the L-chip using the thermocouple 2, the main power voltage level using the voltage test point sensor, and the main power current using the current test point sensor. The cavity was sealed to demonstrate that the PDSM module could be completely integrated into the CROWS. A hole was drilled into the cavity to allow the wireless antenna to be installed on the external surface of the cavity for communications back to the PDSM. A power on reset was installed on the outside of the cavity to repower the PDSM board since it was now sealed. This arrangement was necessary, because the present PDSM board design will intermittently lock up, and the only way to regain functionality of the board is to repower it. The actual cause of this lockup problem has not been determined and requires investigation.

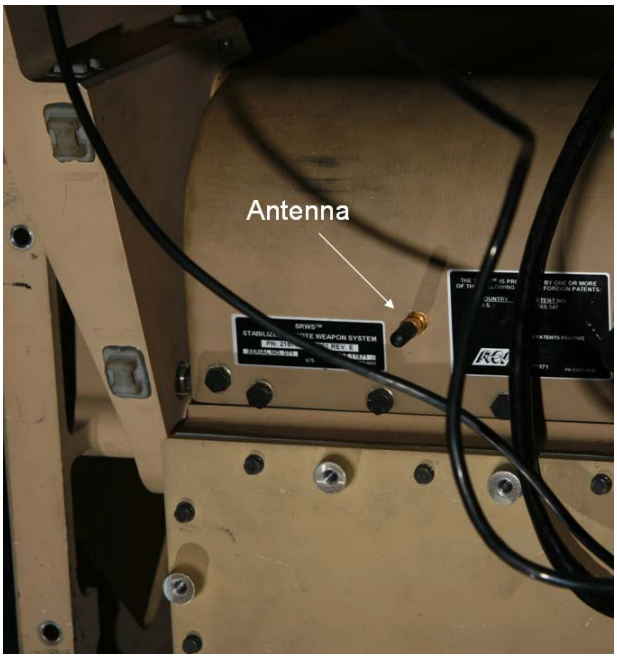


Figure 37. PDSM installed and sealed in the SU motor/actuator cavity.

The PDSM boards successfully monitored the current, voltage, acceleration, and temperature test points within the CROWS, and wirelessly passed the acquired data to the PDSC. The PDSM sensors were successfully programmed to detect and report on test point failures. Lessons learned from installing the PDSM boards into the CROWS were that the process is very time consuming and runs the risk of damaging the CROWS. During the installation process, we broke some CROWS wires that had to be repaired. Also, in one case, the CROWS would not function at all after we installed a PDSM board for an indeterminate reason. After hours of troubleshooting, we never determined the cause of the problem, but eventually, the CROWS became functional. Also, during the installation, the cavities were very tight space-wise, and, in particular, when we sealed the SU motor/actuator cavity we had to be very careful with how wires were routed from

the PDSM to the test points of interest. These remarks indicate that installing a PDHMS into a system will be costly, and proper installation procedures have to be well understood and documented.

13. Recommended Changes to the PDHMS Prototype

Throughout the report, we have made many recommendations on way to enhance the present design. For convenience, this section consolidates most of the recommended changes, enhancements, and design weaknesses. Using an RTOS will very likely eliminate some of these problems. Note: These recommendations are not listed in order of importance, because determining relative importance will depend on the nature of the future work being performed.

- We noticed in the lab that when the M3000 was not physically connected to the PDSM board through the miniature coax connectors, the voltage levels feeding the MSP430 ADC12 were driven above the MSP430's rating of 3.3 V. This problem must be fixed in the next design.
- The jumpers presently allow setting addresses from 0 through 7, providing a maximum of 8 PDSM nodes in the demo system. Theoretically, 65536 of nodes could be supported in the system by either increasing the number of jumpers to 16 or by using some other means to control the firmware.
- A key problem with the present sensor input design is a lack of protection circuitry on the sensor inputs. During use, we damaged several PDSM boards as a result of misconnecting the voltage input on the power supply and voltage sensor input. Using protection circuitry would make the design more robust to inputs that may exceed design limits
- Another concern with the present sensor input design is that using a screw terminal is not an ideal way to connect and remove the sensors from the P&D board. Investigating a better way to do this should be addressed in a redesign.
- When the power is removed from the board, the RTC circuitry loses its previously set time.
- When making the sensor terminal connections, several problems were apparent:
    - The many wires coming off of the board to connect the external sensors can be a problem if the space in which the PDSM board is installed is very tight.
    - The screw terminal connector is not easy to work with, especially when frequently connecting and disconnecting sensors. Some form of a quick release terminal connector should be investigated.

    - It is possible that a sensor may inadvertently disconnect while in use, possibly due to system vibrations. A firmware method should be put in place to automatically detect when a sensor is no longer connected to the PDSM board

- Although CC2420 can support ZigBee, the design does not implement ZigBee. Future development should use the TI ZigBee stack to make the wireless communications more robust, or implement a proprietary protocol to achieve the same results. These changes will need to address mesh networking, automatic route rediscovery, and some form of automatic acknowledgements to eliminate the problem of dropped packets.
- Given the low 250-kbps data rate, there is not enough bandwidth to stream large amounts (GB) of data quickly enough. Another communication standard with a high data rate may be needed if transferring large amounts of data becomes a requirement of the PDHMS application. This design change would effectively eliminate the use of the CC2420 transceiver and the ZigBee protocol as described previously.
- In the firmware, we suggest eliminating the use of global variables and finding a way to communicate this information in another manner.
- The I2C library needs to be enhanced to make it more robust in dealing with the I2C bus communication collisions that can occur when two or more nodes attempt to access the bus at the same time.
- The low-power sleep mode does not work. When the design goes into sleep mode, the PDSM boards hang; therefore, we used polling instead. This area needs to be investigated for future development in order to find a way to conserve power on the design. Using an RTOS will very likely resolve this problem.
- The USB/UART driver library in the present hardware design does not implement hardware handshaking control lines in the USART communications used for the USB interface. Although it may not be needed, implementation of hardware handshaking control lines should be considered to guarantee more robust communications on the USB/USART interface.
- A key problem with the current GUI interface is that it is not scalable, i.e., if many (hundreds of) nodes were added into the system. The GUI should be redesigned to support this concept.
- We must clearly understand what the envisioned data acquisition modes are for the system, because this knowledge dictates what the required processors we will use. We must determine the required accuracies for each sensor as this can significantly affect the hardware design.
- Future implementation should examine the remote sensor configuration more carefully and devise a better approach. One approach could be to use more 32-bit words; another

approach could be to use an XML-based configuration dictionary. There could be many other approaches.

- The data file storage formats must be reviewed more carefully to ensure that the storage formats support the long-term storage requirements of the PDHMS.
- Although we used LEDs for PDSM board status indication, future designs should consider using other indicators that may be more user friendly. For instance, one could use low-power LCD displays to show ACSII status messages and report on sensor measurements without the need to feed that data back to the GUI. This area should be further investigated for future designs.
- We currently implement primarily Cartesian displays for viewing the raw data and its frequency spectrum. The type of displays a user may actually benefit most from in such a system requires further investigation.

14. Future Development

This section describes the present PDHMS system limitations and provides our vision for what a low-power P&D sensor system should actually look like. We discuss these items in the context of modular units that are networked, are capable of reporting system status to GUI/user, and may need increased processing power. This section details what we believe an ultimate PDHMS for monitoring military equipment should look like. We need to do a white paper study examining what others are doing in this field before making any conclusions.

Based on experience gained in this prototype efforts, we have determined that developing a remote PDHMS can become quite complex. Such a system requires various sensor hardware; wired and wireless communications; data storage and reporting; real-time status reporting; hardware multi-tasking; the flexibility to adapt to different measurement and operational environments and insert new processing algorithms; and the ability to manage potentially thousands of networked sensor devices and computing nodes. These requirements suggest the need for a highly flexible operating system to manage the multitude of tasks and a custom or COTS real-time embedded operating system at the heart of the architecture. Choosing such an operating system can be complicated; such an operating system must be chosen with these requirements in mind. Furthermore, a key to the operating system is that a strong digital signal processor (DSP) math library and strong communications I/O drivers must be available for the targeted processors. There are many commercial RTOSs and DSPs that support these requirements.

With regard to sensor, there are many sensor types on the market—acoustics, airflow, current, chemical, electromagnetic, force, humidity, liquid, motion, optical, position, pressure, proximity,

speed, temperature, vibration, and voltage sensors, to name a few. Not all systems that require monitoring will need all of these sensor types. A more sophisticated PDSM design concept should be able to support any desired combination of sensors without requiring a major redesign. A new design architecture should be able to add or remove any subset of the sensors as needed, for instance, a "plug and play" concept where the user could tailor the selection of sensors by plugging them into the PDSM carrier board. This concept requires the PDSM board to support connecting sensor daughter cards onto it. These daughter cards would contain one or more of the sensors and provide the needed sensor conditioning circuitry and, possibly, memory for temporary data storage. Each daughter card would also have a standard bus for connecting onto the PDSM board, which would allow it to transfer data from the sensor board to the PDSM card for storage, processing, or reporting status information to the end user. The primary purpose of the PDSM board would be to provide processing algorithms and the communications capabilities to transfer the data and status reports to the end user through hardwired or wireless communications channels.

As noted throughout this report, there are many design limitations that need to be addressed for such a board to truly support the demands of a sophisticated PDHMS architecture. A primary concern is that, although the MSP430 can do some limited processing, it not intended as a DSP for complex algorithm implementations. It cannot support the more sophisticated processing algorithms demanded by real-time P&D systems. A redesign might either replace or incorporate the MSP430 with a more powerful low-power MCU, such as an ARM or TMS320 DSP. These processors have advanced math libraries available for implementing DSP algorithms.

The prototype GUI and PDHMS, as implemented, have some key limitations. One primary limitation is scalability. If the number of nodes in the system were to increase dramatically, for example, to 100 nodes, this interface would not be able to scale at all as far as how information is presented to the operator or how the operator can configure the system. Further investigation is required to develop a more generic, scalable design.

15. Conclusions

ARL has developed and tested a wireless rudimentary P&D sensor system. We encountered various challenges even in such a simple system; however, designing and implementing such a prototype has given ARL greater insight into how a more sophisticated PDSM should be designed. Various lab measurements and demonstrations were performed with the ARL PDHMS. The ARL PDHMS has been completely documented in this report along with a detailed summary of the systems capabilities and weaknesses. We have also provided recommendations for improving the current design and developing future redesigns. The most notable finding is that a more scalable user interface for the command, control, and configuration for the PDHMS must be investigated.

The P&D program will continue to evolve as ARL partners with the Tank and Automotive Research, Development and Engineering Center (TARDEC) to do work on the Integrated Vehicle Health Monitoring System for Tactical Wheeled Vehicles. This program will further develop the sensors and architecture design, and work towards ruggedizing the PDSM boards for shock, vibration, environment, packaging, sensor connections, and I/O protection circuitry. In addition, high-speed communications will be integrated into the system, controller area network (CAN) bus will be used, and a RTOS will be implemented. Also, we will investigate a DSP, which will likely be integrated with the present design.

16. References

1. "2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver – Datasheet." Texas Instruments Incorporated, 2008. **cc2420.pdf**
2. *±1.5g - 6g Three Axis Low-g, Micromachined Accelerometer*; Freescale Semiconductor Technical Data, Rev 1; Freescale Semiconductor, June 2005. **MMA7260Q-Rev1.pdf**
3. *CSA1V Current Sensor*; Rev. 002; Sentron AG, Switzerland, April 2005. **CSA-1V.pdf**
4. Bierl, L. *Interfacing the 3-V MSP430 to 5-V Circuits*; SLAA148; Texas Instruments, October 2002. **slaa148-msp430-interfacing-to-higher-voltage-circuits.pdf**
5. *LM134/LM234/LM334 3-Terminal Adjustable Current Sources General Description*; DS005697; National Semiconductor, March 2005. **LM334appnotes.pdf**
6. *Current Sensing with the CSA-1V Hall IC AN_102: Operation and application of the Sentron CSA-1V-SO surface mount current sensor*; AN_102; GMW, San Carlos, CA, November 2004. **AN_102_REV_C.pdf**
7. *Implementing a Direct Thermocouple Interface with the MSP430x4xx and ADS1240*; SLAA125A; Texas Instruments, October 2001. **slaa125a.pdf**
8. *Accelerometer, User's Manual*, Rev. 2; Vibra-Metrics Part #9350-1000, Vibra-Metrics, Princeton Jct., NJ, June, 2004. **VibraMetricsAccelerometerUsersManual.pdf**
9. *Interfacing the MSP430 With MMC/SD Flash Memory Cards*, Application Report SLAA281B–November 2005–Revised March 2008. **slaa281b.pdf**

Appendix. CD Directory Structure and Bill of Materials

A-1. CD Directory Structure

This section gives a brief overview of the contents of the companion CD of this report:

- ALTIUM_final-demo-design-with-mods-design-fy-2009 contains the Altium Designer design documents for the PDSM.
- CROWS Demo Photos contains photos of the CROWS system during final demonstration.
- CROWS Demo Video contains videos of the CROWS system during final demonstration.
- Documents contains this document and all relevant technical reference documentation for implementing the PDSM board.
- IAR Embedded Workbench contains the MSP430 firmware for all PDSM peripherals.
- PSPICE Simulations contains simple PSPICE circuit simulations of some the PDSM sensor circuitry.
- CROWS Demo MATLAB contains the data and MATLAB code to view the PDSM data files taken at the final CROWS demonstration.
- USB Driver contains the Windows-based device driver for communicating from the PC to the PDSM board. This is required to be installed before the PDSM GUI will work.
- Visual Studio Projects contains the source code and executable for the PDCS GUI interface.

A-2. Bill of Materials

Table A-1 lists the complete bill of materials for the PDSM board as generated by Altium Designer.

Table A-1. Bill of materials for the PDSM board as generated by Altium Designer.

Bill of Materials

Bill of Materials For PCB Document [PD_crows.PcbDoc]

Source Data From: **PD_crows.PcbDoc**

Project: **PD_crows.PrjPCB**

Variant: **None**

Creation Date: 2/1/2010 3:12:52 PM

Print Date: 40210 40210.64195

Footprint	Comment	LibRef	Designator	Description	Quantity
CAP-T491B	10uF	Cap Pol1	C1, C2, C10	Polarized Capacitor (Radial)	3
CAP-0603	0.1uF	Cap	C3, C5, C11, C16, C18, C19, C20, C45, C48	Capacitor	9
CAP-0603	0.22uF	Cap	C6	Capacitor	1
CAP-0603	0.01uF	Cap	C7, C8	Capacitor	2
CAP-0603	8pF	Cap	C9, C14	Capacitor	2
CAP-0603	12pF	Cap	C15, C17	Capacitor	2
CAP-0402	0.5pF	Cap	C21, C23	Capacitor	2
CAP-0402	5.6pF	Cap	C22, C24, C26	Capacitor	3
CAP-0402	27pF	Cap	C25, C27	Capacitor	2
CAP-0805	0.1uF	Cap	C28, C42	Capacitor	2
CAP-0402	0.1uF	Cap	C29, C30, C31, C32	Capacitor	4
CAP-0402	0.01uF	Cap	C33, C40	Capacitor	2
CAP-0402	68pF	Cap	C34, C35, C36, C37	Capacitor	4
CAP-0805	10uF	Cap	C38, C39	Capacitor	2
CAP-0805	100pF	Cap	C41	Capacitor	1
CAP-0805	2.2uF	Cap	C43	Capacitor	1
CAP-0603	0.33uF	Cap	C44	Capacitor	1
CAP-0603	10uF	Cap	C49	Capacitor	1
LED_SMD	LED	LED	D1, D2, D3	LED	3
DO-35	1N457	1N457	D4, D5, D6	Low Leakage Diode	3
DO-35	1N5221C	1N5221B	D7, D8	Silicon Zener Diode (0.3 to 0.5W)	2

ANT-2.4	2.4GHz Antenna	ANT-2.45	E1		1
ARL_LOGO	ARL LOGO	ARL LOGO	G		1
HDR2X7	Header 7X2	Header 7X2	J1	Header, 7-Pin, Dual row	1
SMA	SMA-F	SMA-F	J3	SMA Female Connector	1
HDR1X2	Header 2	Header 2	JP1, P3	Header, 2-Pin	2
RES-0402	5.6nH	Inductor	L1	Inductor	1
RES-0402	7.5nH	Inductor	L2, L3	Inductor	2
SD_HRS	SD Card Connector Hirose	SD Card Connector Hirose	P1	SD Card Connector Hirose	1
MMCX2.54-V5	COAX-M	COAX-M	P2, P5, P9	RF Coaxial PCB Connector, MMCX; Thru-Hole, Vertical Mount Plug, 50 Ohm Impedance	3
HDR1X3	Header 3	Header 3	P4	Header, 3-Pin	1
TERMSTRIP10	TERMSTRIP10	TERMSTRIP10	P6		1
HDR1X2	10k Thermister	Header 2	P7	Header, 2-Pin	1
HDR2X2	Header 2X2	Header 2X2	P8	Header, 2-Pin, Dual row	1
RES-0603	47K	RESISTOR	R1	Resistor	1
RES-0805	0	RESISTOR	R2, R3	Resistor	2
RES-0805	300	RESISTOR	R4	Resistor	1
RES-0603	1K	RESISTOR	R5, R6, R7	Resistor	3
RES-0603	560	RESISTOR	R8, R9	Resistor	2
RES-0603	3.3M	RESISTOR	R10	Resistor	1
RES-0805	75	RESISTOR	R11	Resistor	1
RES-0603	10K	RESISTOR	R12, R13	Resistor	2
RES-0805	10K	RESISTOR	R14, R15	Resistor	2
RES-0402	43K	RESISTOR	R16	Resistor	1
RES-0805	130K	RESISTOR	R17, R19, R20, R21, R22, R23	Resistor	6
RES-0805	2	RESISTOR	R18	Resistor	1
RES-0805	44K	RESISTOR	R24	Resistor	1
RES-0805	4K	RESISTOR	R25, R26, R27	Resistor	3
RES-0603	67	RESISTOR	R28, R32, R36	Resistor	3
RES-0603	670	RESISTOR	R29, R33, R37	Resistor	3
RES-0603	9.5K	RESISTOR	R30, R34, R38	Resistor	3
RES-0603	2.5k	RESISTOR	R31, R35, R39	Resistor	3
RES-0603	2K	RESISTOR	R40	Resistor	1
PBSW_SMD	SW-PB	SW-PB	S1	Switch	1
SO-G8/X.6	LT1521_S	LT1521_S	U1	300mA low dropout regulator with shutdown	1
D2PAK	L7824CD2T	L7824CD2T	U2	24V Positive Voltage Regulator	1

SSO-G28/E4.3	ADS1241	ADS1241	U3		1
F-QFP10x10-G64/P.5N	MSP430F1611	MSP430F169	U4	TI 16 Bit MicroController	1
QFN16 1MM	MMA7260Q	MMA7260Q	U5	Freescale 3-Axis Accelerometer	1
QLP48	CC2420	CC2420	U6	Chipcon RF Transceiver	1
182H_N	REF1004C1.2	REF1004C1.2	U7	1.2V and 2.5V Micropower Voltage Reference	1
QFN16 0.5mm	M41T62	M41T62	U8	STMicro Real Time Clock	1
M14A_L	LP324M	LP324M	U9	Micropower Quad Operational Amplifier	1
SO8_N	LM334D	LM334D	U10, U11, U12	Three Terminal Adjustable Current Source	3
XTAL SMD 2x2.4	8MHz	XTAL_SMD	X1	Crystal	1
XTAL SMD 3.2x1.5	32KHz	XTAL TF	X2, X4	Crystal TF	2
XTAL SMD UM	16MHz	XTAL	X3	CSX3-AA-1816.000	1

Bibliography

1.2V and 2.5V Micropower Voltage Reference; REF1004; Burr Brown, 2008. **ref1004-1.2.pdf**

1N5221B - 1N5267B 500mW Epitaxial Zener Diode; DS18006 Rev. 15-2; Diodes Incorporated. **ds18006.pdf**

3M Card Connector SD Normal Polarization, Push-Push, Surfacemount; TS-2198-01; 3M Electronics, Austin, TX, 28 November 2006. **SDcardConnector-3M.pdf**

ADS1240, ADS1241: 24-bit Analog-to-Digital Converter; SBAS173A; Burr-Brown Products from Texas Instruments, June 2001. **ads1241.pdf**

Application Basics for the MSP430 14-Bit ADC; SLAA046; Texas Instruments, June 1999. **slaa046.pdf**

Architecture and Function of the MSP430 14-Bit ADC; SLAA045; Texas Instruments, June 1999. **slaa045.pdf**

Biasing Internally Amplified Accelerometers; SLAP3, Application Note 3.0; Spectral Dynamics, Inc., 1995–2002. **BiasingInternallyAmplifiedAcclerometers-AppNote.pdf**

Carter, B.; Brown, T. *Handbook of Operational Amplifier Applications*; SBOA092A; Texas Instruments, October 2001. **operational-ampfliers-sboa092a.pdf**

Corson, D. W. *Comparing 8-bit microcontrollers for ultra-low power applications*, Low Power Design, October 2005. **someMCUcpmparisons.pdf**

CP2102 Single-chip USB TO UART Bridge; Rev. 1.3, 8/08; Silicon Laboratories, Austin, TX, 2008. **cp2102.pdf**

Datasheet, Vibra-Metrics Model 3000, Miniature Tri-axial Accelerometer, MISTRAS Group Inc., 2009. **Model 3000 Series.pdf**

Ergen, S. C. *ZigBee/IEEE 802.15.4 Summary*, Berkeley University, 10 September 2004. **zigbee.pdf**

Foust, F. *Secure Digital Card Interface for the MSP430*, Dept. of Electrical and Computer Engineering, Michigan State University, 2004. **sdcard_to_msp430_appnote_foust.pdf**

"IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks Specific Requirements, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)." IEEE Computer Society, The Institute of Electrical and Electronics Engineers, Inc., New York, 2003. **802.15.4-2006.pdf**

Judd, J. E. Basics of Acceleration Measurements! Mechanical Failure Prevention Technology, 59th MFPT Forum, 12 April 2005. **ACCELERATION MEASUREMENTS SESSION 4-19-05_comp.pdf**

"Keystone Miniature Sensors 1°C Accuracy." Thermistor. **RL0503 Series,pdf**

L7800 Series Positive Voltage Regulators; Doc ID 2143 Rev 21; STMicroelectronics, March 2010. **L78L24.pdf**

LM134-LM234-LM334 Three Terminal Adjustable Current Sources; SGS-Tomson Electronics, March 1994. **LM334.pdf**

LM9076 BMA-3.3, 150mA Ultra-Low Quiescent Current LDO Regulator with Delayed Reset Output; 200830; National Semiconductor, 12 November 2007. **LM9076BMA-3.3.pdf**

LM9076 BMA-5.0, 150mA Ultra-Low Quiescent Current LDO Regulator with Delayed Reset Output; 200830; National Semiconductor, 12 November 2007. **LM9076BMA-5.0.pdf**

LP324/LP2902 Micropower Quad Operational Amplifier; DS008562; National Semiconductor, July 2001. **LP324.pdf***MSP430f241X, MSP430f261X, Mixed Signal Microcontroller*; SLAS541F; Texas Instruments, June 2007 (revised December 2009). **msp430f2619.pdf**

M*41T93 Serial SPI bus RTC with battery switchover*, Rev 4; STMicroelectrics, August 2008. **M41T93.pdf**

Mitchell, G.; Conn, M.; Harris, R.; Bayba, A. *Automated Data Acquisition for a Prognostics and Diagnostics Health Monitoring System*; ARL-TR-4523; U.S. Army Research Laboratory: Adelphi, MD, July 2008. **p&d_whitepaper1.pdf**

MSP430F261x/241x Device Erratasheet; SLAZ033F; Texas Instruments, October 2007 (revised January 2010). **slaz033f.pdf**

MSP430x2xx Family User's Guide; SLAU144E; Texas Instruments, 2008. **slau144e.pdf**

MSP-FET430 Flash Emulation Tool (FET) (for Use With Code Composer Essentials for MSP430 Version 3.1) User's Guide; SLAU157H; Texas Instruments, May 2005 (revised November 2008). **slau157h.pdf**

Omega Temperature Measurement Handbook, 6th ed.; Omega Engineering Incorporated, 2007. **z019-020.pdf**

Predicting the Battery Life and Data Retention Period of NVRAMs; ST AN1012; STMicroelectronics, May 2001. **m41t93-app-notes.pdf**

Revised Thermocouple Reference Tables, TYPE-K Reference Tables; NIST Monograph 175 Revised to ITS-90; NIST. **z218-220.pdf**

SanDisk MultiMediaCard and, Reduced-Size MultiMediaCard, Product Manual, Version 1.0 Document No. 80-36-00320, May 2004. **SanDiskSDcardManual-rs-mmcv1.0.pdf**

SanDisk Secure Digital Card, Product Manual, Version 1.9; Document No. 80-13-00169; San Disk, Milpitas, CA, December 2003. **SanDiskProdManualSDCardv1.9.pdf**

SD Card Specification Simplified Version of: Part E1 Secure Digital Input/Output (SDIO) Card Specification, Version 1.00; San Disk, Milpitas, CA, October 2001, SD Association, **SD_SDIO_specsv1.pdf**

Single-Chip USB to UART Bridge; DS014-1.0, Preliminary, Cygnal, August 2003. **usb-tranceiver-356495_1.pdf**

"SLAA281B–November 2005–Revised March 2008, Application Report, Interfacing the MSP430 With MMC/SD Flash Memory Cards." Texas Instruments Incorporated**.** **SLAA281B.pdf**

ST L78xx L78xxC Positive voltage regulators; Rev. 19; STMicroelectronics, 2008. **L7824CD2T-voltRegulatorl7805.pdf**

Transient Voltage Suppression Diode Arrays, Littlefuse. **SP0502BA.pdf**

Type K Thermocouple, thermoelectric voltage as a function of temperature (°C); reference junctions at 0 °C, Pyromation, Inc. **emfk_c.pdf**

Type K Thermocouple, thermoelectric voltage as a function of temperature (°F); reference junctions at 0 °C, Pyromation, Inc. **emfk_f.pdf**

"Type MS, Epoxy Coated Thermistor" in *NTC Epoxy Chip Series Thermometrics Thermistors*; 920-322A; GE, 2006. **Thermister.pdf**

List of Symbols, Abbreviations, and Acronyms

ADC	analog to digital converter
API	application programmer interface
ARL	U.S. Army Research Laboratory
CAN	controller area network
CBM	Condition Based Maintenance
COTS	commercial of the shelf devices
CROWS	Combat Remotely Operated Weapons System
DMA	direct memory access
DSP	digital signal processor
DSSS	direct sequence spread spectrum
FCF	frame control field
FCS	frame check sequence
FFT	Fast Fourier Transform
FIFO	first-in-first-out
GUI	graphical user interface
IC	integrated circuit
I/O	input/output
LCD	liquid crystal display
LED	light emitting diode
MCU	microcontroller unit
MMC	multimedia card
MPDU	MAC protocol data unit
P&D	Prognostics and Diagnostics
PANs	personal area networks

PDCS	Prognostics and Diagnostics Control Station
PDHMS	Prognostics and Diagnostics Health Monitoring System
PDSM	Prognostics and Diagnostics Sensor Modules
PPTC	Polymer Positive Temperature Coefficient
RAM	random access memory
RMS	root mean squared
RTC	real time clock
RTOS	real time operating system
RX	receive
SCL	serial clock
SD	secure digital
SDA	serial data
SPI	serial peripheral interface
SU	sensor unit
TARDEC	Tank and Automotive Research, Development and Engineering Center
TI	Texas Instrument
TX	transmit
UART	universal asynchronous receiver-transmitter
USART	universal synchronous/asynchronous receiver/transmitter
USB	universal serial bus

No. of Copies	Organization
1 ELEC	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1 CD	OFC OF THE SECY OF DEFNS ATTN ODDRE (R&AT) THE PENTAGON WASHINGTON DC 20301-3080
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV & ENGRG CTR ARMAMENT ENGRG & TECHNLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
2	US ARMY RDECOM-TARDEC CONDITION BASED MAINTENANCE PROGRAMS ATTN AMSRD TAR R G SMITH ATTN AMSRD TAR R K FISCHER 6501 E 11 MILE RD MS 204 WARREN MI 48397
1	PM TIMS, PROFILER (MMS-P) AN/TMQ-52 ATTN B GRIFFIES BUILDING 563 FT MONMOUTH NJ 07703
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD A RIVERA FT HUACHUCA AZ 85613-5300
1	COMMANDER US ARMY RDECOM ATTN AMSRD AMR W C MCCORKLE 5400 FOWLER RD REDSTONE ARSENAL AL 35898-5000
1	US ARMY RDECOM-ARDEC ATTN RDAR WSF A G GARCIA BLDG 91 PICATINNY NJ 07806
1	US ARMY RDECOM-ARDEC ATTN RDAR WSF A D MARSTON BLDG 61S PICATINNY ARSENAL NJ 07806-5000
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
6	DIRECTOR OF ENGINEERING CURTISS-WRIGHT CONTROLS INC ELECTRONIC SYS ATTN A CARTER ATTN A KOTHARI ATTN B PUSZKARCZUK ATTN I PAZ ATTN P MALCHODI ATTN T LOGRASSO 151 TAYLOR STR LITTLETON MA 01460
1	US ARMY RSRCH LAB ATTN RDRL CIM G T LANDFRIED BLDG 4600 ABERDEEN PROVING GROUND MD 21005-5066
14	US ARMY RSRCH LAB ATTN IMNE ALC HRR MAIL & RECORDS MGMT ATTN RDRL CIM L TECHL LIB ATTN RDRL CIM P TECHL PUB ATTN RDRL SER E A BAYBA ATTN RDRL SER E C LY ATTN RDRL SER E D WASHINGTON ATTN RDRL SER E G MITCHELL ATTN RDRL SER E K TOM ATTN RDRL SER E R DEL ROSARIO ATTN RDRL SER M D WIKNER ATTN RDRL SER M E ADLER ATTN RDRL SER M M CONN ATTN RDRL SER M R HARRIS ATTN RDRL SER P AMIRTHARAJ ADELPHI MD 20783-1197
TOTAL:	62 (1 ELEC, 31 CD, 30 HCS)