System Level Integration of Large Area Electronic Sensor Arrays for the Purpose of Structural Health Monitoring

Sam Payne

May 14, 2013

Naveen Verma

Submitted in partial fulfillment of requirements for the degree of Bachelor of Science in Engineering Department of Electrical Engineering Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

[illegible]

Sam Payne

System Level Integration of Large Area Electronic Sensor Arrays for the Purpose of Structural Health Monitoring

Sam Payne

ABSTRACT

As infrastructure continues to age, Structural Health Monitoring provides a valuable resource for tracking the integrity of buildings, bridges, tunnels, and pipelines. Current Structural Health Monitoring solutions have proven expensive and unreliable. Large Area Electronics (LAE) provide affordable, flexible, and reliable arrays of sensors capable of returning the necessary information to monitor the health of structures. This project aims to improve an already existing Large Area Electronics system to provide low-power and low-cost system level capabilities. A library of functions written for an MSP430 microcontroller provides the system with low-level functions to integrate with ICs that interface with LAE sensor arrays. From these primitive measurement and communication functions, a larger system can be constructed using the flexible MSP430 family. This report explains the function of this library and presents the results of interfacing and testing the controller's functionality.

ACKNOWLEDGMENTS

The development of this research and document took place under the supervision and guidance of Naveen Verma. Without his supervision, this project could not have even gotten off the ground. The project team involved in the development of this system included Warren Rieutort-Louis, Liechao Huang, Yingzhe Hu, and Josue Sanz-Robinson, headed by professors Sigurd Wagner, James C. Sturm, and Naveen Verma. Without their contributions, the system would not exist, and this project could not have been completed. Yingzhe Hu served as an invaluable resource, helping to establish proper communication between electronic components in this project. Yingzhe also provided the necessary testing equipment setup and background information. Warren Rieutort-Louis served as an excellent educator, providing introductory information and many details about the system. Warren also aided in the development of the interface circuits used for testing. Princeton’s School of Engineering and Applied Science provided the necessary funding for this project.

Table of Contents

1 Introduction

Structural Health Monitoring (SHM) continues to be a more important issue as infrastructure ages. Current SHM solutions are expensive and unreliable. However, Large Area Electronics (LAE) provides a method of creating reliable and cheap sensor arrays capable of providing the necessary tools for SHM. Naveen Verma's LAE team at Princeton University has been working to develop an SHM system using LAE. This project expands on that system to provide basic functions to control the system and expand isolated sensor arrays into a full network.

2 Motivation

As infrastructure ages, tunnels, bridges, buildings, and other structures tend to develop imperfections. When left unchecked, these imperfections can lead to losses, failures, and disasters. Structural Health Monitoring (SHM) systems aim to detect imperfections in structures before they become problems. SHM can provide early warnings about warping, strain, and cracking in structures so that maintenance and repairs can be planned in advance. However, SHM is rarely utilized in structures today. There are two reasons behind the lack of SHM implementation: cost and reliability. Both of these problems prohibit investment in SHM from being worthwhile. Current solutions use sensors made from expensive materials. Due to the expense of these sensors, they are often placed sparsely and therefore they can only provide low resolution data. Often, readouts from these systems do not reliably warn about problems in structures. However, these problems in cost and resolution can both be overcome by employing Large Area Electronics (LAE) systems.

LAE systems use electronics that are printed on thin film polymers and provide functionality to components through amorphous silicon. This technology allows for the creation of densely populated arrays of sensors at low cost. This overcomes both the cost and accuracy troubles that plague current SHM solutions.

3 Background

3.1 Structural Health Monitoring

As mentioned above, structural health monitoring is becoming more important given the aging infrastructure of the United States and around the globe. It is important to monitor structural integrity of bridges, tunnels, and buildings to minimize risks of disaster. In spite of this issue's importance, SHM systems are seldom used due to a lack of reliable and affordable solutions. Current solutions provide sparse sensor networks, which cannot provide a full picture of structural risks. The first signs of deterioration are cracks, which cannot be detected unless they occur close to sensor locations. It has been demonstrated that sensor networks with sparsely spaced nodes will often fail to detect cracks [1, 2].

Professors Naveen Verma and Branko Glisic present two excellent examples of why sparse detection systems are not suitable for reliably detecting structural damage. Streicker Bridge on Princeton University's campus has served as a test site for numerous SHM methods. Long-gauge sensors embedded in the Streiker Bridge while pouring concrete failed to detect cracks that were more than a meter away [1]. When these cracks were detected at less than a meter, measurement readouts were often difficult to diagnose since the change in strain was comparable to changes in strain caused by temperature variation [1]. It is important to note that the sensors in contact with the damage were the only sensors to display a change in readout. More surprisingly, these sensors would have still detected the crack even if their sensitivity had been an order of magnitude lower. This application demonstrates the importance of direct damage detection over attempts to detect damage at a distance, and it also reveals how unreliable point sensors can be even at very high sensitivity levels.

Buried concrete pipelines serve as another notable case study by Glisic. In his experiment, ground movement caused crushing of joints in a pipeline equipped with distributed fiber-optic sensors [2]. The crushing was successfully detected due to a high strain change at the direct location of the damage; however, sensors less than 50 cm away registered only bending without damage to the pipe. Again, we note that the sensors that detected the fault could have been an order of magnitude less sensitive and still would have detected the structural damage. This again shows the importance of direct damage detection for the purposes of reliability and sensitivity.

Building a dense two-dimensional array or using the same sensors would be prohibitively expensive. Classic resistive strain gauges, some of the cheapest strain sensors, cost about $20 each [3]. Building a high density array of these sensors is predicted to cost around $32,000 per square meter, not including the cost of bonding, wiring, and use of such a system [3]. Though such a system would provide more reliable direct damage detection, the cost of such a system is prohibitive. Therefore, an alternative technology should be used to build such high density arrays. Large Area Electronics provides the solution to this problem.

3.2 Large Area Electronics

Large Area Electronics (LAE) is an emerging technology that allows for deposition or printing of thin-films of electronic components on plastic sheets. Components are made from amorphous silicon, which allows for flexible placement on a variety of surfaces. Additionally, the components are not bound by the cost of crystalline silicon; they can therefore be produced at larger scales at a lower cost factor. Thin-film transducers, including pressure sensors, vapor sensors and particle sensors have been demonstrated using this technology [4]. These sensors can also be produced in dense arrays for high-resolution sensing.

Additionally, the integration of thin film transistors (TFT) has been demonstrated in LAE. These transistors cannot provide the same high performance of transistors made in crystalline silicon due to the nature of large area substrates. However, the transistors can be used to provide the basic functionality of reading out large arrays of sensors [3]. Therefore, since TFTs cannot provide all the computational functionality required for such an array, circuits created from conventional silicon ICs must interface with the TFTs. These ICs provide the remaining required functions to read out measurements from these arrays. For these reasons, LAE is an excellent solution for providing high density sensors for SHM. The system this project attempts to improve (described below) takes advantage of an LAE sensor array to provide SHM solutions.

Current systems using large sensing arrays of point sensors often use wireless communication between nodes. Wireless communication can be costly given the amount of power required for these technologies to function. Fortunately, LAE provides low-power communication options. Leads printed on thin films can provide long distance, wired communication, and low-power readout from large arrays of sensors. It has been shown that power can be saved in significant portions by technologies that use wired communication rather than wireless. For an SHM application, it is important to keep power costs down for the sake of scalability. To combat high power usage, solutions for obtaining power through solar power harvesting systems have also been demonstrated on LAE systems [7]. To summarize, LAE SHM system are capable of providing high density, reliable sensor data while remaining self-powered. These flexible systems could also be tailored in form and scale to fit various applications.

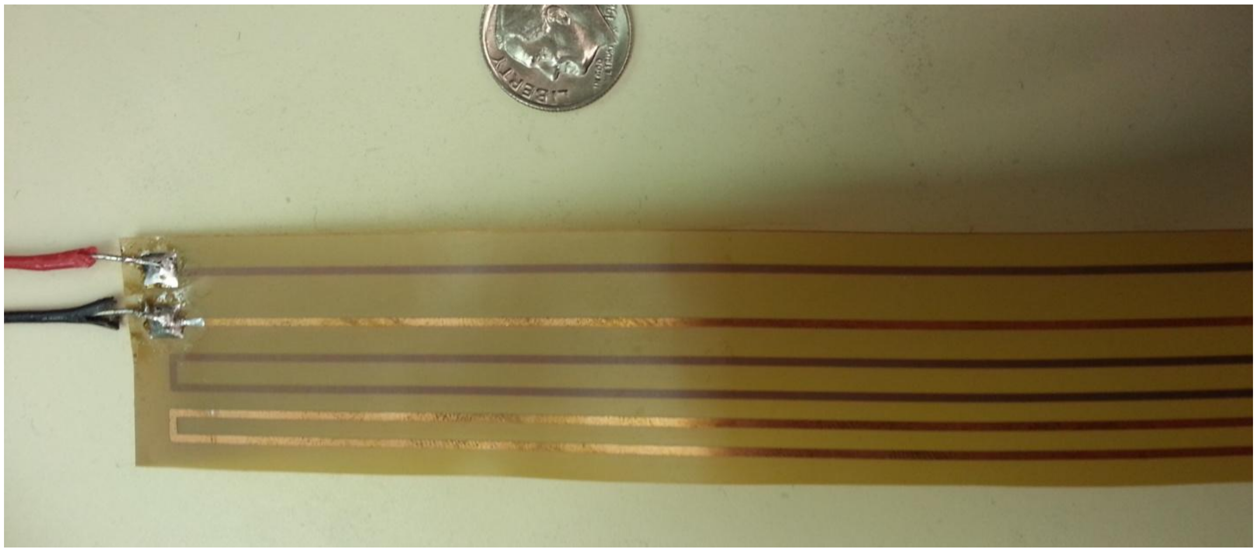

Figure 1: Wired communication substrate printed on thin film

4 Sensor Array System

As referenced above, the system this project aims to expand has two main components – one based on crystalline silicon ICs, the other on flexible semiconductor films. Components including sensors, sensor access, energy-harvesting, and communication are all present on the thin film. Sensors are distributed in an array across the sheet and are coupled with TFTs to provide basic readout functions. The flexible film also houses large area interconnects so that ICs can communicate with one another. The ICs provide all interface necessities to use the sheet to initialize measurements, read measurements, and communicate with one another.

Traditionally, IC to LAE coupling is difficult. Reliable physical connections are hard to create and pose a large limitation on scalability. For this system, non-contact couplings have been used to overcome this dilemma. The current system uses short range, inductive, and capacitive antennas patterned on both the LAE sheet and on the IC. The two antennas interact at close proximity to induce current in one another at a low energy cost. TFTs located on the LAE sheet provide enough functionality that very few signals are required to be sent over this interconnect. This allows for full power-efficient control of the system through reliable IC to LAE coupling.

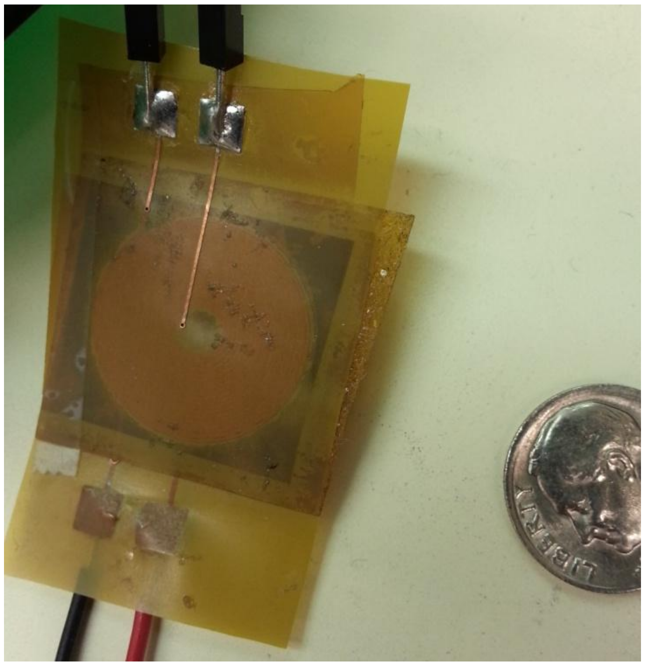
Figure 2: Inductive interconnects printed on thin film for easy coupling of LAE to interfacing ICs

As mentioned, communication between ICs is achieved through a large area interconnect. However, this channel is extremely long and thus holds unpredictable impedance, which will have a large effect on the signal-to-noise ratio. To avoid problems in transmission, transmitting ICs are calibrated to send signals using the resonant frequency of the interconnect. By using on-off keying at this resonant frequency, a higher SNR can be achieved, allowing for more reliable, lower-power communication.

Equipped with the ability to send all necessary signals to the LAE sheet, the interfacing ICs only require signals for calibration and activation. This project aims to create a library of functions capable of delivering the proper signals to these ICs to allow basic operations to be carried out. Once this library is in place, higher system level integration can be put into place.

5 Project Goals

The goal of the project is to develop a library of functions for a microcontroller to control the ICs that interface with the LAE sensor array. This library should be capable of sending the required signals to these chips to allow measurement and readout from the dense array of sensors. The second purpose of the library is to communicate between ICs. The microcontroller will be responsible for initiating communication and allowing ICs to transmit and receive over a wired bus printed on the same film surface on which the sensors are located. This library allows for basic readout and storage of measurements for coordination with a larger system. The library also allows for synchronization of communication between arrays of sensors so that the system can be flexibly adjusted to match power saving needs. This includes studying the system with which these microcontrollers will interface, selecting a platform on which to develop (see the "Platform" section), development of signals based on requirements (see "Measurement" and "Communication" sections), and testing these signals with the prototype system.

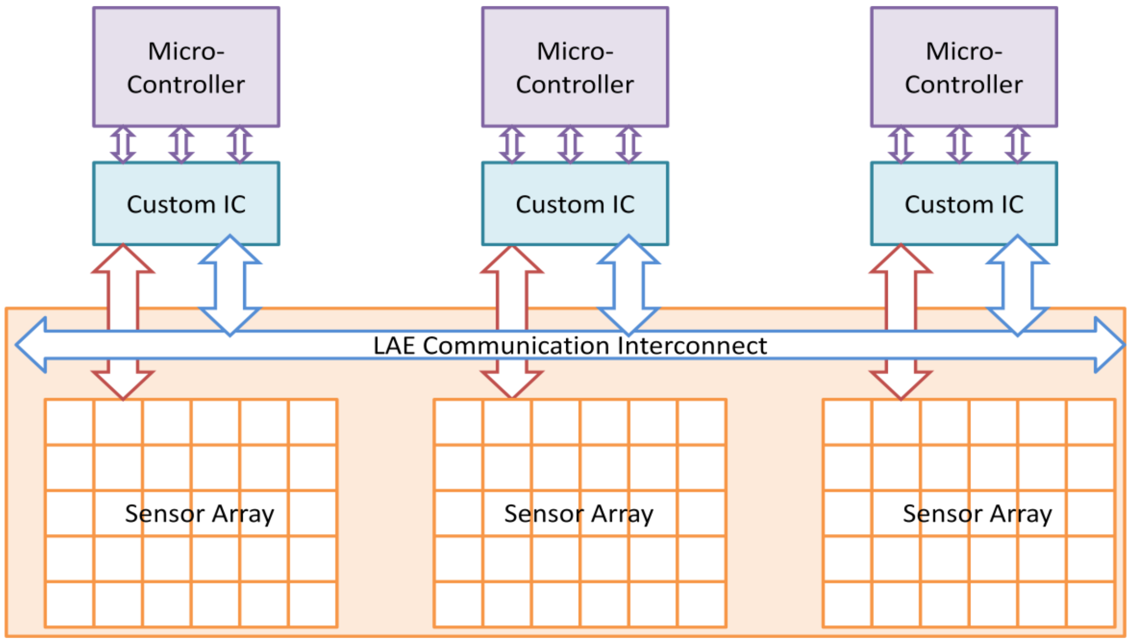

Figure 3: a high-level view of the LAE system with microcontrollers

6 System Design

6.1 Platform

The chipset to interface with the LAE system was chosen with special attention paid to on chip resources, power consumption, cost, and available development support. Since the full system will be self-powered, we want to keep power consumption at a minimum. Therefore, the platform we are developing on should use low-power microcontrollers. Since the system will be highly scalable for larger structures, these microcontrollers should be cheap. An expensive platform would defeat the purpose of the project. Finally, we would like to enable a larger network structure to be built on top of these sensors. Future steps in the project should allow compatibility with networking standards and protocols that have already been developed.

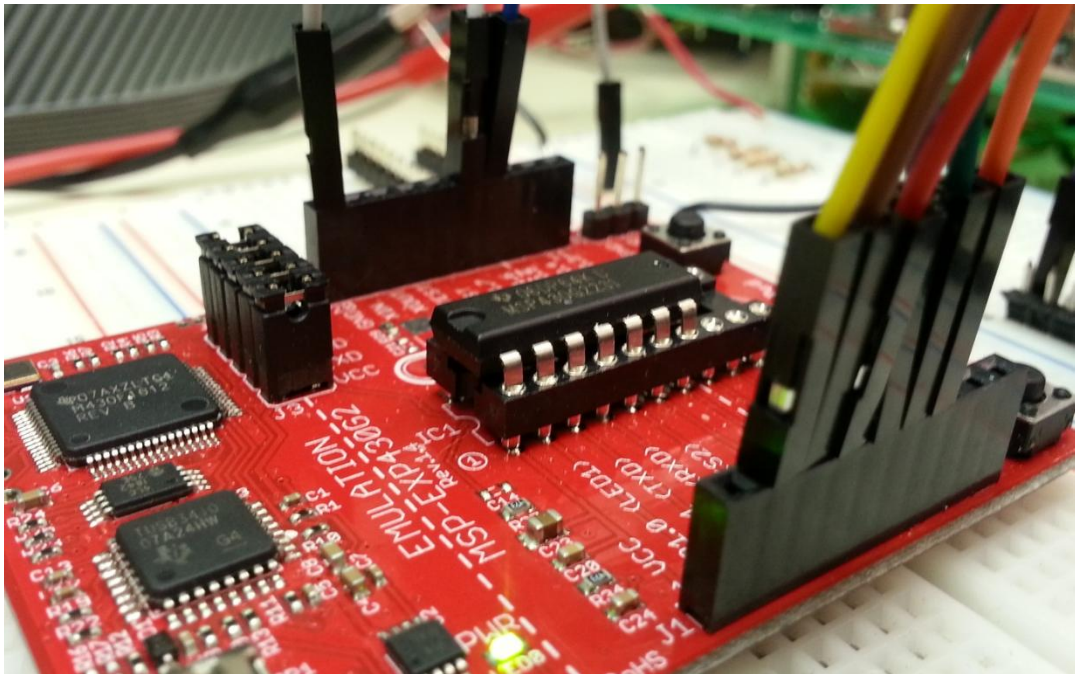

Figure 4: The 14 pin MSP430-G2231 microcontroller used in this project

The MSP430 family of processors provides low-cost, power-efficient chips that are capable of wired and wireless networking. MSP430 processors are available and widely used in industry, and they have a wide range of supporting software. The development environment for this chip is free, easy to use, and convenient. Code written for MSP430s can be written to be compatible with other processors in the MSP430 family, which gives us flexibility in future system builds. Each MSP430 controller can run in several low-power modes for power saving in higher level software, each enabling fewer resources for power-saving purposes. More information about these operation modes can be found in the MSP430 product brochure [5]. MSP430 devices are also compatible with a variety of peripherals, including some designed specifically for remote sensor networks [5]. Finally, MSP430 models can be run from a variety of clock sources, allowing for lower power use. The prototype for this project is an MSP430G2231 model; this processor possesses the necessary resources to execute this library while providing low-cost and low-power use. The controller runs off a 1Mhz clock, but can be sourced from a variety of other clock sources up to 16 Mhz.

6.2 Overview of the Single Timer System

The library is written to require only one timer on chip and is primarily interrupt bound. This allows other on chip timers to remain free for higher level computation. Running only one timer also allows for low-power consumption, but it limits the number of operations that can be carried out simultaneously. The designed software system consists of three primary states: an idle state, a measurement state, and a communication state. These states are exclusive; to enter either the measurement state or the communication state requires seizing of the timer. In the idle state, the timer is freed for other high level applications.

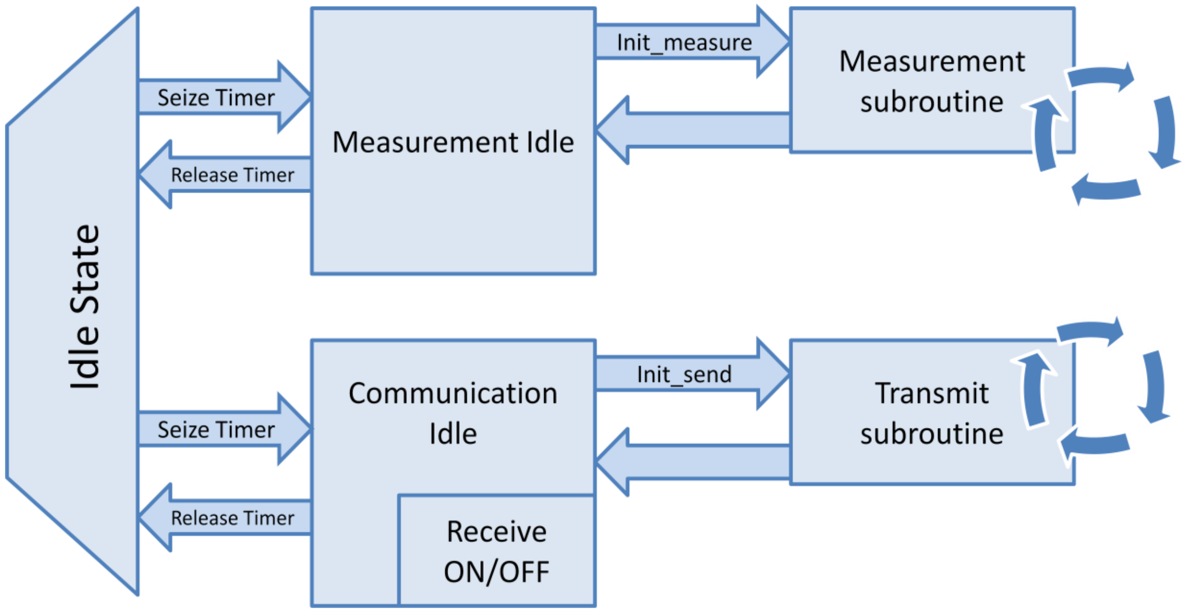

Figure 5: Illustration of single timer system states

While the system is in either the communication or measurement state, the functionality of these states is interrupt bound. Therefore, higher level functions can continue to run on the CPU while lower level communication with the IC runs in the background. Only when higher level functions need the timer are they required to turn off the communication or measurement systems. Since the timer counts by the CPU's clock frequency, to maintain consistent timing characteristics, counting variables must be adjusted linearly for different clock frequencies. The library's timing characteristics can easily be linearly adjusted for this purpose.

6.3 Measurement System

The first part of the library of functions designed for the microcontroller provides the correct signals to the ICs to initiate and read out measurements from the sensor array. Since much readout functionality is provided by TFTs on the LAE sheet, the IC only needs to send four signals over the inductive interconnect to communicate with the array. While the IC operates at 1.2 volts, the LAE circuits need over 6 volts for reasonable performance. The inductive interfaces use AC-modulated signals to communicate so they can simultaneously provide voltage step-up [7]. Amorphous Silicon Schottky thin-film diodes with low voltage drop and good rectification characteristics have been designed and printed on the LAE sheet to demodulate the signals sent by the IC [4]. The IC sends a 3-phase scan control signal with SCAN 1-3 asserted in round-robin order to cycle through each sensor in the array [4].

The controller must be capable of commanding the IC to initiate the scan and terminate the scan. Measurements are read from the array, in order, and temporarily stored in the IC for the controller to read out until overwritten by the next measurement. The controller provides a clock signal to the IC to read out measurements taken during the scan. The controller is also equipped with sleep and wake operations in order to allow greater power optimization, and to forfeit the timer for use by the communication system or higher level code to use. Shown below is the API for the measurement system:

`int init_scan();`	Sends global reset signal to can chain, starts scan clock to run the scan chain Error -1: Scan chain already active
`int terminate_scan();`	Stops the scan clock Error -1: Scan clock not running
`int wake_measurment();`	Wakes measurement system – will not allow transmitter or receiver to wake at the same time Error -1: Transmission system is active
`int init_meas( unsigned int* buffer);`	Takes a measurement, reading out to value pointed to by “buffer.” This function runs in the background. Scan clock must be running for this to occur Error -1: Scan clock not running

Figure 6: Measurement system API

The init scan function will first send a 6ms pulse as a reset signal to the IC. This reset signal is followed by the scan clock sent on a different pin. All pins for both the measurement and communication systems are selectable and can be adjusted as the user sees fit. The scan clock has a period of 3ms, half that of the reset signal. All parameters given are specific to this calibration and can be adjusted by the user if desired. The scan clock will drive the three phase readout signal which alternates in round-robin order. The scan clock can be left to run indefinitely without reading out any measurements.

The init_meas() function begins a subroutine that will return the next measurement taken into the provided buffer. Since this entire process is interrupt bound, the controller actuates signals, then sets a timer to wake up again in the next state. Each transition is triggered by the timer. Below is a diagram of the state transitions that occur as this measurement subroutine runs in the background.

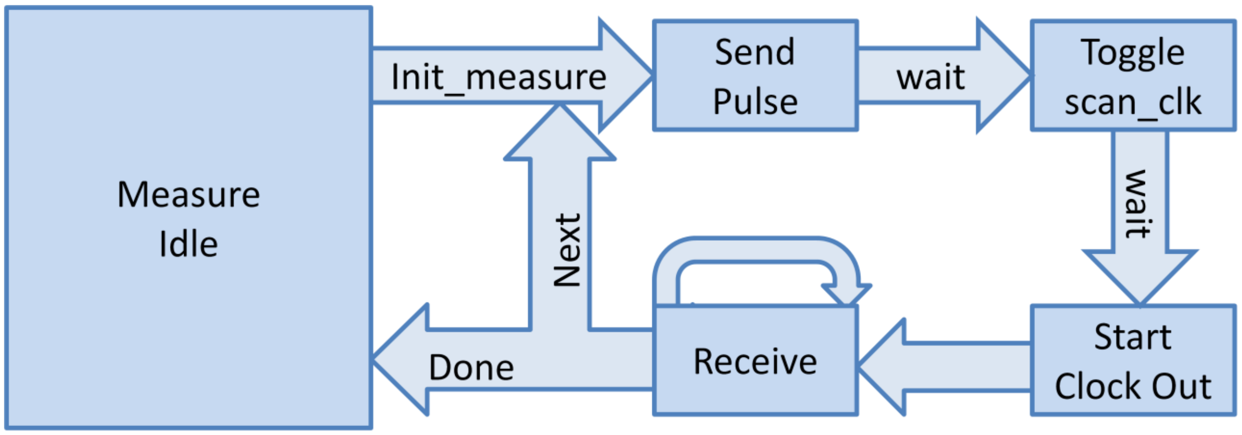

Figure 7: An illustration of the measurement subroutine

The initial pulse sent is a pulse to initialize measurement in the IC. Once the measurement is started, a measurement capacitor is charged over a time period (set by an external resistor) that integrates to determine the magnitude of the readout from the sensor. This integration process must finish before the readout clock attempts to read the digital value of the sensor. During this integration process, it may be necessary to switch the scan clock. Since this whole process is interrupt bound, a separate state must be added to keep the scan clock alternating at the correct frequency.

Once the integration has been given enough time to complete, the readout clock is started. In our testing procedures, the frequency of the readout clock is equal to the maximum speed at which the MSP430 was able to read out data from chip. In an MSP430 with a higher clock frequency, this time could be reduced. As always, the user can select the frequency by changing parameters in the header-file. The clock receives 16 bits before the measurement is completed. Once the measurement is complete, the next state is determined by observing if more measurements are waiting to read out. Measurement calls will block until the current measurement is complete.

During the next integration period, higher level software can easily extract received data from the buffer or assign a new buffer since it will remain untouched until the receive clock turns on again. Below is a capture of each of the signals named above (excluding the global reset pulse):

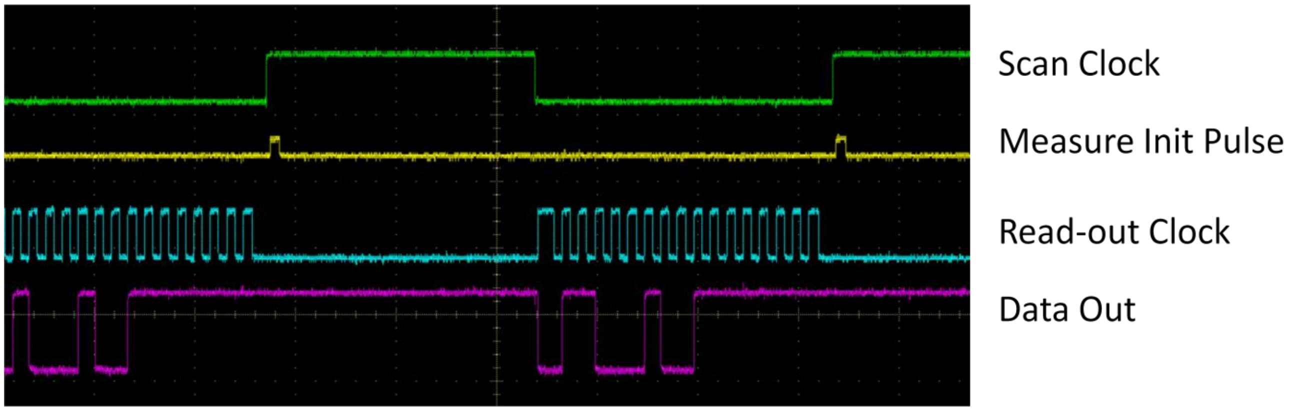

Figure 8: The three main control signals for measurement initialization and readout

One can easily observe a full cycle of the measurement subroutine beginning with the first rising edge of the scan clock (in green). The measurement initialization pulse is sent (in yellow) followed by the falling edge of the scan clock after an integration period. The read-out clock is then started (in blue), which extracts the data sent from the IC (in purple). Finally, another measurement is triggered which will repeat the same cycle, reading from the next sensor in the array.

6.4 Communication System

Each IC is capable of acting as a transceiver on the communication lines printed on the LAE film. The microcontroller is responsible for initiating these communications and aiding the IC in receiving messages. Each IC contains a transmitting and receiving unit. The microcontroller is capable of sleeping and waking the communication system as a whole for power-saving purposes, or to forfeit the timer. The microcontroller is also capable of feeding data to the transmission unit to be sent. The clock signal for this data is extracted from the signal. The microcontroller also supplies a clock to the IC to read out values on the data bus. Since each microcontroller sends its own system clock signal to the interfacing circuit's receiving unit, the receiving controller's clock may have a different phase from the sender. Therefore, receiving units must be able to align their clock with the message being sent. The microcontroller provides the ability to shift the receiving clock signal incrementally for use in a higher level communication synchronization subroutine. Below is the API for this system:

`int start_comm();`	Will turn on communication system, seizing timer0 Error -1: Measurement system is on
`int stop_comm();`	Will turn off the communication system, freeing timer 0 Error -1: communication already off

`int start_tx_send(`
`   unsigned int* buffer);`
	Sends the contents of "buffer" over the transmit channel. 'buffer' will not be altered. Will return error if a send is already in progress.
	Error -1: the transmit clock is not on
	Error -2: the transmit system is busy – send already in progress

`int shift_rx_clk();`	Will shift the rx clock by one "div" cycle to synchronize with transmitter (selectable depending on frequency) Error -1: the rx clock is not running
`int stop_rx_clk();`	Will turn off the rx clk. Error -1: rx clock is already off

`int alternate_rx_rcv_ON(`
`   unsigned int* buffer1,`
`   unsigned int* buffer2,`
`   int leng);`
	This function allows the user to alternate between two receive buffers to prevent overwriting of one while reading.
	'buffer1' is written into first and 'buffer2' is written into second. The 'leng' parameter specifies how many bits should be read before alternating. This allows for convenient payload reading.
	Error -1: rx clock is not on

`int alternate_rx_rcv_OFF(`
`   unsigned int* buffer);`
	This function turns off alternating receiving. The rx function will now only read into a single 'buffer.'
	Error -1: rx clock is not on
	Error -2: alternate receiving mode is not active

Figure 9: API for communication system

Once the communication system is turned on, messages can be transmitted. The transmit function sends a 16 bit value (at a specified frequency) to the IC, which extracts a clock from the signal and sends the 16 bit message over the bus. The transmit function will return an error if another transmission is in progress, but it will not block to wait for that transmission (this must be done by the user, if desired). The transmit function will not block program flow while transmission occurs since it is interrupt bound.

Once the communication system is turned on, the receiving system can also be activated. The receiving system can function in two modes: single buffer mode and alternating buffer mode. The single buffer mode will constantly fill a 16 bit buffer with data from the receive readout of the IC. This readout mode is ideal if a controller is listening for its ID, or a “gold code” to appear on the bus. The alternating buffer system, once activated, will fill one buffer with a given number of bits, then switch to another buffer and fill that buffer with the same number before alternating back. This allows for easy readout of sequential data blocks sent across the bus. The alternating receive function requires two buffers and the length before switching buffers to be specified. Turning either of these systems on will immediately start feeding a clock to the IC to read out data.

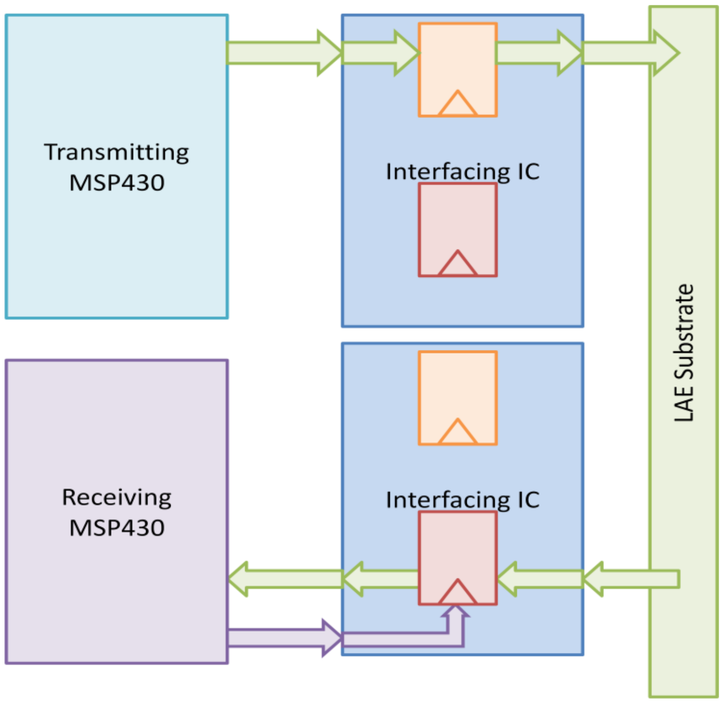

Figure 10: Illustration of communication between two microcontrollers via LAE interconnect

Since two ICs could not be connected for testing, the transmission signals were tested on a single IC. The chip would send a code to itself over LAE interconnect. Upon receiving this code, it would send a response to itself followed by a packet of data. Below we can observe each communication signal – the transmit data in yellow, received data in blue, and receive clock in green. This image shows the transmission of the header, then the response.

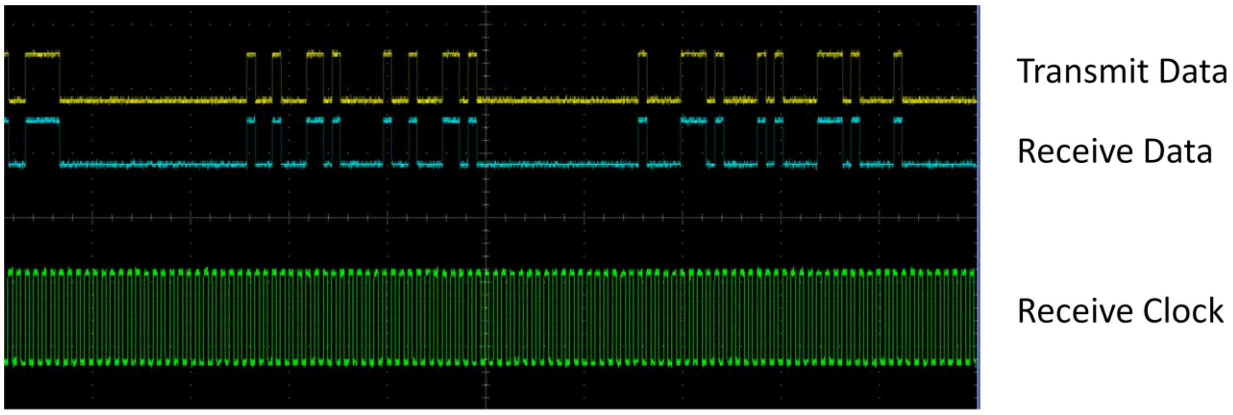

Figure 11: Communication testing on a single IC

On a single controller, the clock signal supplied for readout and the clock signal supplied for transmission are the same clock signal. The clock frequency is determined by two variables: COMM_F and COMM_DIV. The product of these two parameters is equal to half the period of the clock signal (in microseconds, if a 1Mhz clock is used). The COMM_F parameter serves as a base multiplier, while the COMM_DIV parameter determines how far the communication clock is shifted in one shift. To align clock signals with the transmitting unit, the shift_rx_clk function is used.

The shifting function extends the period of one cycle by 1/20 of the original period, returning to the original frequency immediately after (1/20 is determined by the COMM_DIV parameter which can be set by the user). The shifting function requires 10 periods between shifts since continuous calling of this function without forced delay can expand the length of one cycle indefinitely. The shifting function could not be tested using two ICs since the equipment was not available. Since an IC communicating with itself will have an aligned clock, we used a signal generator and oscilloscope to confirm that phase shifting of the receiving clock was indeed occurring.

Below we can observe the receive clock (yellow) compared to a reference clock (blue) generated via signal generator. The next figure illustrates the delay between shifts, which occurs consistently when the shift function is constantly called.

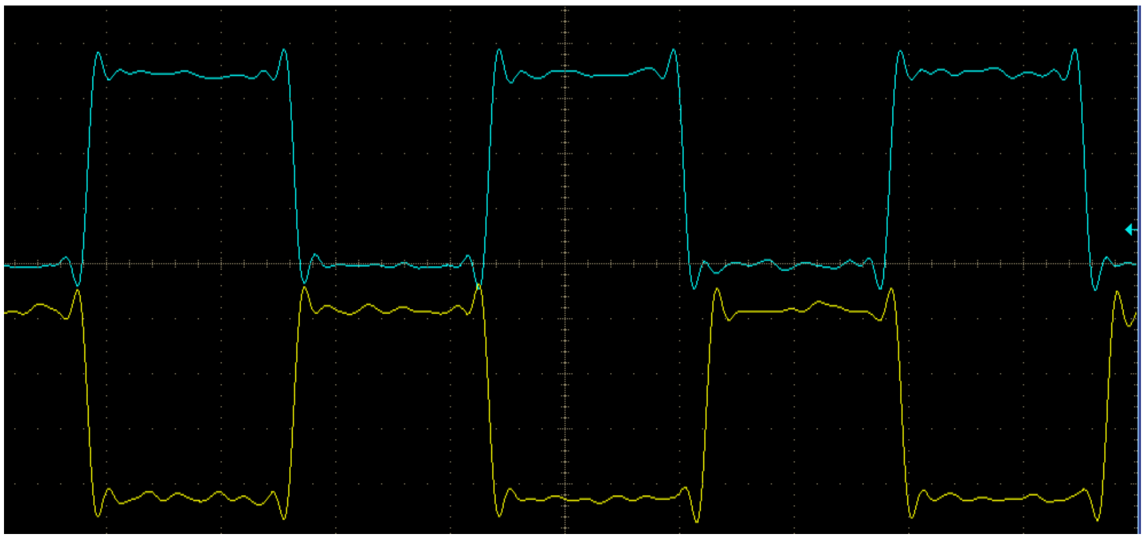

Figure 12: Illustration of a single shift (1/20th of the period)

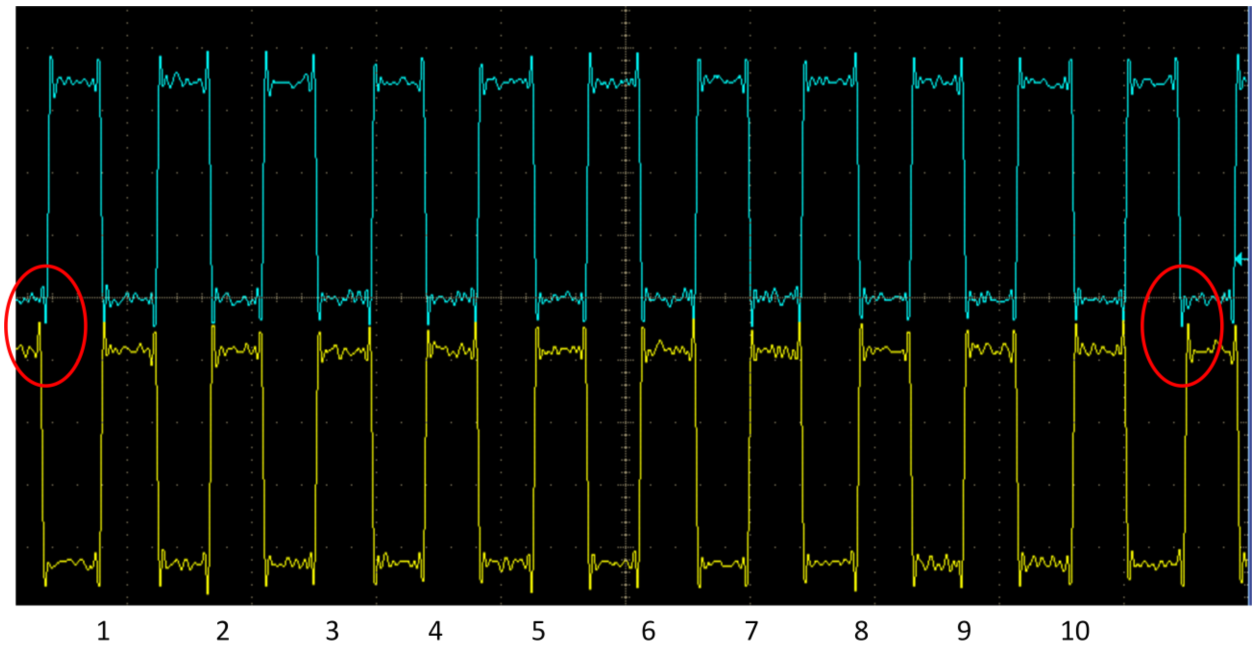

Figure 13: Illustration of delay between shifts. Phase shifts are labeled in red.

These signals provide the foundation for a higher-level network to be put into place. With communication protocols in place, chips will be capable of reading out measurements, sending data to other nodes in the network, commanding other nodes to sleep for a period of time, and many other useful functions.

6.5 System Interface

Since the MSP430, powered at 3.3 volts, sends and receives logical high and low values at 3.3 volts, an interface board had to be prepared to allow communication between the IC and microcontroller. The schematic for this interface board is shown below. Other controllers in the MSP430 family can be powered at a range of values, including 1.2 volts.

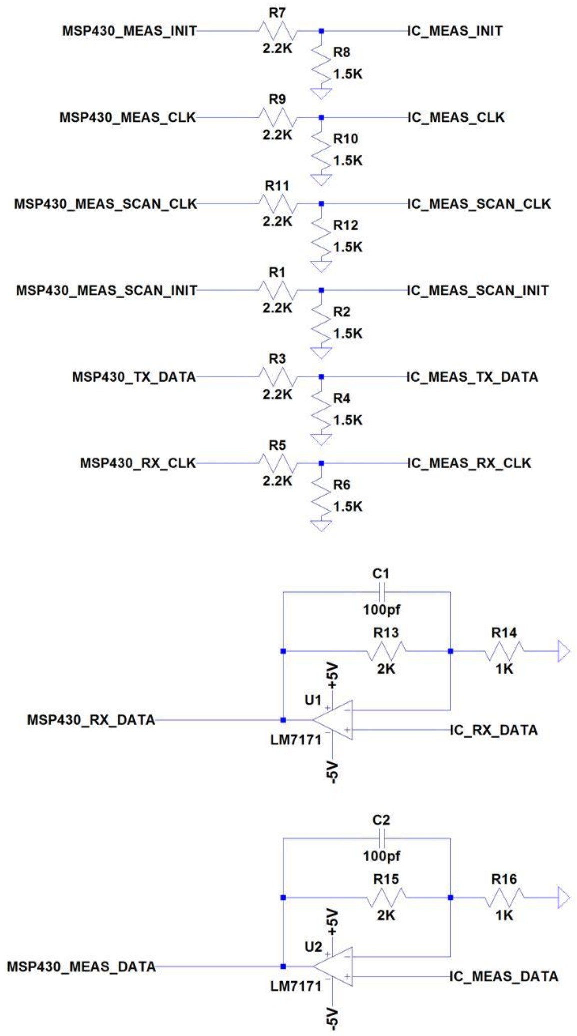

Figure 14: Design of interfacing boards between the MSP430 microcontroller and dedicated LAE interface IC

7 Conclusions and Next Steps

This project successfully implemented signal generating functions to provide measurement readout and network level integration to a large area electronics sensor network. Specifically, using an MSP430 microcontroller, signals can be generated to initiate measurement readout, transmission, and receiving of data. The API established allows for flexible, low-cost and low-power use while leaving ample resources, including the CPU, for other computation by high-level code. This is only a step in making a fully functional sensor network capable of monitoring the health of a structure. The next step of this project will be to establish a network protocol and functions by which microcontrollers can communicate with one another. This section describes future steps in fleshing out this system.

7.1 Future Platform Selection

As mentioned before, the MSP430 family offers a variety of platforms and peripherals for different applications. This section aims to describe a number of controllers and devices compatible with the library developed in this project which could be used to develop a larger system. More information about these platforms can be found in the MSP430 Product Brochure [5].

The MSP430 family of devices offers a variety of devices which can fit this application including the F1xx, G2xx, and F2xx models. The L092 model provides excellent power savings with enough GPIO pins and memory to run the library developed in this application. The device is limited to a 4MHz clock and can be powered as low as .9 volts (.9-1.65). While the L092 controller can serve as a convenient readout peripheral, the CC430 controllers supply powerful functionality, which can be used for larger network coordination. These controllers are compatible with most peripherals and each possesses a high number of GPIO pins capable of coordinating many nodes and peripherals. Notably, the CC430 also provides low-power RF functionality designed for remote sensing applications.

Energy harvesting development kits also exist for MSP430 devices. Since these devices use such low-power, they can be self-powered, which is ideal for many structural health monitoring applications. The eZ430-RF2500-SHE is a complete energy harvesting development kit available from Texas Instruments [5]. As mentioned before, the CC430 controller supplies wireless functionality, and shorter-range Bluetooth access is also available for MSP430 devices with the addition of the CC2560 Bluetooth platform [6]. All of these peripherals and specialized controllers can be tailored to create a larger network ideal for low-power SHM using LAE sensors.

7.2 High Level Communication

As mentioned before, the library of functions in this project create a foundation on which higher level software can be built. An important step in the development of a larger network includes the establishment of communication between microcontrollers on the same LAE sheet using the send, receive, and shift clock functions. Some primitive code for a suggested protocol has been sketched out and included in the appendix, we will aim to describe the functionality of this protocol here. First, the sending controller will broadcast a destination code over the LAE interconnect periodically, and wait for a response from the receiver. Each other controller listens to the interconnect to detect if it is the intended destination. Since the clocks of the transmitting and receiving controller may not be aligned, the receiving controller should shift its clock periodically until it detects its unique destination code. When the receiver has detected its code, the clocks of the transmitter and receiver are aligned. The receiver sends a response to the transmitter to signal that it is prepared to receive data.

The transmitter then sends packets of data in the following manner: Each packet should contain a 16 bit head and tail. The receiving controller will listen in single buffer mode until it receives the 16 bit head of the packet. When the head is received, the receiver switches to double-buffer mode, receiving 16 bit chunks of data and storing each for later use by higher level code. Each 16 bit chunk of data is checked to be the tail, and when the tail is received, bit checking is performed to ensure that the packet received is intact. A final confirmation is sent back to the transmitter by the receiver; more packets are sent or the communication is ended. This protocol assumes that each controller shares the same data bus and that higher level software can prevent cross-talk on this bus.

7.3 Final Remarks

A strong foundation has been put into place by this project for development of a larger network. With controllers capable of reading out measurements from the LAE sensors, a larger SHM system can be built to reliably monitor structural integrity at a lower cost. Hopefully future development can build on this project to demonstrate the full utility of this kind of system.

References

1. Glisic, B. 2011. Streicker Bridge: an on-site SHM laboratory at Princeton University campus. SMAR, Paper No. 306, Dubai, UAE.
2. Glisic, B, and Oberste0Ufer, K. 2011. Validation Testing of Fiber Optic Method for Buried Pipelines Health Assessment after Earthquake-Induced Ground Movement, Proceedings of 2011 NSF Engineering Research and Innovation Conference, Atlanta, GA, USA.
3. Glisic, B, Verma, N. 2011. Sensing Sheet for SHM Based on Large Area Electronics. 5th International Conference on Structural Health Monitoring of Intelligent Infrastructure (SHMIII-5). 2011. Cancun, Mexico.
4. Y. Hu, W. Rieutort-Louis, J. Sanz-Robinson, K. Song, J. Sturm, S. Wagner, and N. Verma, "High-resolution Sensing Sheet for Structural-health Monitoring via Scalable Interfacing of Flexible Electronics with High-performance ICs," *VLSI Symp. Circuits (VLSI)*, June 2012.
5. "MSP430™ Ultra-Low-Power Microcontrollers Product Brocure" Texas Instruments Incorporated. Dallas, Texas. 2013. [http://www.ti.com/lit/sg/slab034v/slab034v.pdf]
6. "MSP430™ + CC2560 Bluetooth® Platform Product Brocure," Texas Instruments Incorporated. Dallas, Texas. 2010. [http://www.ti.com/pdfs/wtbu/SWPT038.pdf]
7. Y. Hu, W. Rieutort-Louis, L. Huang, J. Sanz-Robinson, S. Wagner, J. Sturm, and N. Verma, "Flexible Solar-Energy Harvesting System on Plastic with Thin-film LC Oscillators Operating Above f_t for Inductively-coupled

Appendix

Appendix 1: Header File

```
#ifndef __yingzheIC
#define __yingzheIC

int setup();
int init_scan();
int terminate_scan();
int wake_measurment();
void sleep_measurment();
int init_meas(unsigned int* buffer);

int start_comm();
int stop_comm();
//int start_tx_clk();
int start_tx_send(unsigned int* buffer);
//int stop_tx_clk();
int start_rx_clk(unsigned int* buffer);
int shift_rx_clk();
int stop_rx_clk();
int alternate_rx_rcv_ON(unsigned int* buffer1, unsigned int* buffer2, int leng);
int alternate_rx_rcv_OFF(unsigned int* buffer);


#endif
```

Appendix 2: Library Functions and Interrupt Handlers

```
#include <msp430g2231.h>
#include "yingzheIC.h"

//Constant variables
enum var_state {AWAKE, ASLEEP, SHIFTING};

//Measurment Ports
const int MEAS_INIT_PORT      = 0; //OUT  sends out a pulse to initialize measurment
const int MEAS_IN_PORT        = 1; //IN   reads in measurment
const int MEAS_CLOCK_OUT_PORT = 2; //OUT  output clock for measurment
const int MEAS_AWAKE_PORT     = 8; //---  determines if measurment functionality is on or not
const int MEAS_SCAN_CLK_PORT  = 4; //OUT  scan clock for scan chain
const int MEAS_SCAN_INIT_PORT = 5; //OUT  sends pulse to initialize scan

//Measurment Constants - All constants are configured for a 1Mhz clock (100 = 100 us)
const int SCAN_INIT_WIDTH    = 6000;    //global reset length to start scan chain
const int SCAN_CLK_PERIOD    = 3000;    //half of the scan clock period
const int MEAS_INIT_WIDTH    = 100;     //measurment initialization signal
const int MEAS_WAITING1_TIME = 2900;    //time (before scanclock toggle) to wait for
integration
const int MEAS_WAITING2_TIME = 2900;    //time (AFTER  scanclock toggle) to wait for
integration
const int READOUT_CLK_PERIOD = 50;      //NOT IMPLEMENTED - readout clock currently set to
minimum  period

//Measurment Variables
enum measure_state {INITIATING, MEASURING, WAITING1, WAITING2, READING, IDLE};
int MEASURE_READ_STATUS  = IDLE;
int MEASURE_SCAN_INIT    = IDLE;
int MEASURE_SCAN_STATUS  = ASLEEP;
int MEASURE_EDGE_COUNT   = 0;
int PENDING_MEAS              = 0;
unsigned int* MEAS_READ_OUT_BUFFER;

//Communicaiton Ports
const int TX_CLK_PORT   = 1; //---  send transmitting clock
const int TX_SEND_PORT  = 3; //OUT  send transmitting data
const int RX_CLK_PORT   = 6; //OUT  send receiving clock
const int RX_RCV_PORT   = 7; //IN   reveive receiving data

//Communication Constants - All constants are configured for a 1Mhz clock (100 = 100 us)
const int COMM_F   = 200;  // COMM_F * COMM_DIV = receive period
const int COMM_DIV =  10;  // number of possible shifts to phase align receive clock

//Communication Variables
int RX_CLK_STATE     = ASLEEP;
//int TX_CLK_STATE     = ASLEEP;
int TX_SEND_STATE     = ASLEEP;
int COMM_DIV_COUNT   = 0;
int COMM_STATE       = ASLEEP;
int TX_SEND_COUNT    = 0;
unsigned int* RX_READ_OUT_BUFFER;
unsigned int* ALT_RX_READ_OUT_BUFFER;
unsigned int TX_SEND_BUFFER;
int RX_SHIFT_DELAY   = 0;
int RX_RCV_COUNTER   = 0;
int RX_RCV_LEN       = -1;
```

```
/***************************************************************************************
 * Begin Misc. Functions
 **************************************************************************************/
int setup() {
	P1OUT = 0;
	return 0;
}
/***************************************************************************************
 * Begin Measurment Functions
 **************************************************************************************/

//sends global reset to scan chain starts the scan clock to run the scan chain
int init_scan() {
	//must turn scan off before turning back on
	if(	MEASURE_SCAN_STATUS == AWAKE) return -1;
	MEASURE_SCAN_STATUS = AWAKE;
	MEASURE_SCAN_INIT = INITIATING;

	//set up scan clock
	TA0CCR0 = SCAN_INIT_WIDTH;		// init pulse = 4 ms
	TA0CCR1 = SCAN_CLK_PERIOD;		// scan clk   = 2 ms
	TA0CTL = TASSEL_2 + MC_1;		// Timer A 0 with ACLK @ 1MHz, count UP
	TA0CCTL0 |= BIT4;			// Enable counter interrupts

	P1OUT |= (1 << MEAS_SCAN_CLK_PORT);  //turn on clock port
	P1OUT |= (1 << MEAS_SCAN_INIT_PORT); //turn on initializer port

	return 0;
}

//stops the scan clock
int terminate_scan() {
	if(MEASURE_SCAN_STATUS == ASLEEP) return -1;
	while(MEASURE_READ_STATUS != IDLE);  //wait for current measurment to be done

	MEASURE_SCAN_STATUS = ASLEEP;
	P1OUT &= ~(1 << MEAS_SCAN_CLK_PORT); //turn port off
	return 0;
}

//enters measurment state
int wake_measurment() {
	if(COMM_STATE == AWAKE) return -1;

	//assign appropriate ports for measure
	P1DIR =  ((1 << MEAS_INIT_PORT)
			+ (1 << MEAS_CLOCK_OUT_PORT)
			+ (1 << MEAS_AWAKE_PORT)
			+ (1 << MEAS_SCAN_CLK_PORT)
			+ (1 << MEAS_SCAN_INIT_PORT));

	P1OUT = (1 << MEAS_AWAKE_PORT);

	return 0;
```

```
}

int init_meas(unsigned int* buffer) {
	if(MEASURE_SCAN_STATUS != AWAKE) return -1;

	MEAS_READ_OUT_BUFFER = buffer;

	PENDING_MEAS = 1;
	while(MEASURE_READ_STATUS != IDLE || MEASURE_SCAN_INIT != IDLE) {
		//wait for currennt measurment to finish
		//wait for initialization to finish
	}
	PENDING_MEAS = 0;

	MEASURE_READ_STATUS = INITIATING;
	return 0;
}

/* init_readout_p1 will read in values from an IC, providing
 * a clock signal at clock_port */
int init_readout_p1() {
	P1OUT |= (1 << MEAS_CLOCK_OUT_PORT);		//set pin output to be lo
	*MEAS_READ_OUT_BUFFER = 0;				//clear readout buffer

	//16 cycles output clock handled in inturrupts
	TA0CCR0 = READOUT_CLK_PERIOD;				// Clock Period
	TA0CTL = TASSEL_2 + MC_1;					// Timer A 0 with ACLK @
1MHz, count UP
	TA0CCTL0 |= BIT4;						// Enable counter
interrupts

	return 0;
}
/*****************************************************************************
 * Begin Communication Functions
 *****************************************************************************/
int start_comm() {
	if(MEASURE_SCAN_STATUS == AWAKE) return -1;
	COMM_STATE = AWAKE;

	TA0CCR0 = COMM_F;					// Clock Period
	TA0CTL = TASSEL_2 + MC_1;			// Timer A 0 with ACLK @ 1MHz, count UP
	TA0CCTL0 |= BIT4;					// Enable counter interrupts

	return 0;
}

int stop_comm() {
	if( COMM_STATE != AWAKE) return -1;
	COMM_STATE = ASLEEP;

	TA0CCTL0 &= ~BIT4;					// Disable counter interrupts

	return 0;
}
/*****************************************************************************
 * Begin Transmit Functions
 *****************************************************************************/
/*int start_tx_clk() {
	if( COMM_STATE != AWAKE) return -1;
```

```
        P1DIR &= ~(1 << TX_CLK_PORT);         //set pin to input - pin is not used
        P1DIR |= (1 << TX_SEND_PORT);         //set pin to output

        TX_CLK_STATE = AWAKE;

        return 0;
}*/

int start_tx_send(unsigned int* buffer) {
        if( COMM_STATE != AWAKE) return -1;          //communication systems are not on
        if(TX_SEND_STATE == AWAKE) return -2;        //current send is in progress

        P1DIR &= ~(1 << TX_CLK_PORT);         //set pin to input - pin is not used
        P1DIR |= (1 << TX_SEND_PORT);         //set pin to output

        TX_SEND_BUFFER = *buffer;
        TX_SEND_STATE = AWAKE;
        return 0;
}

/*int stop_tx_clk() {
        while(TX_SEND_STATE == AWAKE);               //wait for current transmission to finish
        P1OUT &= ~(1 << TX_CLK_PORT);                //ensure that output of port is zero
        TX_CLK_STATE = ASLEEP;
        return 0;
}*/

/*************************************************************************************
 * Begin Receive Functions
 *************************************************************************************/
int start_rx_clk(unsigned int* buffer) {
        if(MEASURE_SCAN_STATUS == AWAKE) return -1; //only one clock can be active at a time
        P1DIR |= (1 << RX_CLK_PORT);                //set pin to output

        RX_READ_OUT_BUFFER = buffer;

        RX_CLK_STATE = AWAKE;
        return 0;
}

int shift_rx_clk() {
        if(RX_CLK_STATE != AWAKE) return -1;
        if(RX_SHIFT_DELAY < 20) return -2; //cannot shift clock too often

        RX_CLK_STATE = SHIFTING;
        RX_SHIFT_DELAY = 0;

        return 0;
}

//function for easy payload extraction
int alternate_rx_rcv_ON(unsigned int* buffer1, unsigned int* buffer2, int leng) {
        if(RX_CLK_STATE != AWAKE) return -1;

        RX_READ_OUT_BUFFER = buffer1;
        ALT_RX_READ_OUT_BUFFER = buffer2;
        RX_RCV_LEN = leng;
        RX_RCV_COUNTER = 0;

        return 0;
}
```

```
int alternate_rx_rcv_OFF(unsigned int* buffer) {
	if(RX_CLK_STATE != AWAKE) return -1;
	if(RX_RCV_LEN == -1) return -2;

	RX_READ_OUT_BUFFER = buffer;

	RX_RCV_LEN = -1;

	return 0;
}

int stop_rx_clk() {
	if(RX_CLK_STATE != AWAKE) return -1;
	P1OUT &= ~(1 << RX_CLK_PORT);			//ensure that output of port is zero
	RX_RCV_LEN = -1;					//turn off buffer alternation
	RX_CLK_STATE = ASLEEP;
	return 0;
}
/*************************************************************************************
 * Begin high level communication functions
 ************************************************************************************/


/*************************************************************************************
 * Begin interrupts
 ************************************************************************************/

/*
 * timer A0 is used for measurments and for communication
 */
#pragma vector=TIMER0_A0_VECTOR
   __interrupt void Timer0_A0 (void) {			// Timer0 A0 interrupt service routine

  /*************************************************************************************
   * Communication Inturrupts
   ************************************************************************************/
	  if(COMM_STATE == AWAKE) {
		COMM_DIV_COUNT++;

		if(RX_CLK_STATE == SHIFTING) { //this will result in a shift by one div unit
			COMM_DIV_COUNT--;
			RX_CLK_STATE = AWAKE;
		}

		if(COMM_DIV_COUNT == COMM_DIV) {
			COMM_DIV_COUNT = 0;
			RX_SHIFT_DELAY++;

			if(RX_CLK_STATE == AWAKE) {
				P1OUT ^= (1 << RX_CLK_PORT);

				if(P1OUT & (1 << RX_CLK_PORT)) { //only measure after rising
edge
					*RX_READ_OUT_BUFFER = (*RX_READ_OUT_BUFFER << 1);
					*RX_READ_OUT_BUFFER += ( ( (P1IN & (1 << RX_RCV_PORT) )
								>> RX_RCV_PORT) );
					RX_RCV_COUNTER++;

					//swap buffers if neccessary
```

```
                                        if( (RX_RCV_LEN != -1) && (RX_RCV_COUNTER ==
RX_RCV_LEN) ) {
                                                //swap out buffers
                                                unsigned int* temp = RX_READ_OUT_BUFFER;
                                                RX_READ_OUT_BUFFER = ALT_RX_READ_OUT_BUFFER;
                                                ALT_RX_READ_OUT_BUFFER = temp;

                                                RX_RCV_COUNTER = 0; //reset counter
                                        }
                                }
                        }


                        if(TX_SEND_STATE == AWAKE) {
                                P1OUT ^= (1 << TX_CLK_PORT);

                                if(P1OUT & (1 << TX_CLK_PORT) ) {
                                        unsigned int msb = TX_SEND_BUFFER & 0x8000;
                                        //send most significant bit of buffer
                                        if (msb)
                                                P1OUT |=  ( 1 << TX_SEND_PORT );
                                        else
                                                P1OUT &= ~( 1 << TX_SEND_PORT );

                                        TX_SEND_COUNT++;
                                        TX_SEND_BUFFER = TX_SEND_BUFFER << 1;

                                        if(TX_SEND_COUNT == 16) {   //sending is finished
                                                TX_SEND_COUNT = 0;           //reset send count
                                                TX_SEND_STATE = ASLEEP;  //signal send function
to return
                                        }
                                }
                        }
                }

                return;
        }
   /*******************************************************************************
    * Measurment Inturrupts
    ******************************************************************************/
        if(MEASURE_SCAN_STATUS == AWAKE && MEASURE_SCAN_INIT == IDLE ) {

                if(MEASURE_READ_STATUS == IDLE) {
                        P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);       //flip scan clock
                        TA0CCR0 = SCAN_CLK_PERIOD;
                        TA0CTL = TASSEL_2 + MC_1;                 // Timer A 0 with ACLK @
1MHz, count UP
                }
                else if(MEASURE_READ_STATUS == INITIATING ) { //Begin measurment initiation
if scan clk low
                        if( P1OUT & (1 << MEAS_SCAN_CLK_PORT)) {  //if scan clock high, wait
for low
                                P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);       //flip scan clock
                                TA0CCR0 = SCAN_CLK_PERIOD;
                                TA0CTL = TASSEL_2 + MC_1;  // Timer A 0 with ACLK @ 1MHz,
count UP
                                return;
                        }

                        MEASURE_READ_STATUS = MEASURING;
			P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);	//flip scan clock to high

			TA0CCR0 = MEAS_INIT_WIDTH;		//Timer counter limit set to
time
			TA0CTL = TASSEL_2 + MC_1;		// Timer A 0 with ACLK @
1MHz, count UP
			TA0CCTL0 |= BIT4;			// Enable counter interrupts

			P1OUT |= (1 << MEAS_INIT_PORT);		//set pin output to be hi
		}
		else if(MEASURE_READ_STATUS == MEASURING) { //end measurment initiation
			MEASURE_READ_STATUS = WAITING1;
			P1OUT &= ~(1 << MEAS_INIT_PORT);	//toggle measurment
initiation port

			//wait until scan clk toggle
			TA0CCR0 = MEAS_WAITING2_TIME;		//Timer counter limit set to
time
			TA0CTL = TASSEL_2 + MC_1;		// Timer A 0 with ACLK @
1MHz, count UP
			TA0CCTL0 |= BIT4;			// Enable counter interrupts
		}
		else if (MEASURE_READ_STATUS == WAITING1) {
			MEASURE_READ_STATUS = WAITING2;

			TA0CCR0 = MEAS_WAITING2_TIME;		//Timer counter limit set to
time
			TA0CTL = TASSEL_2 + MC_1;		// Timer A 0 with ACLK @
1MHz, count UP
			TA0CCTL0 |= BIT4;			// Enable counter interrupts

		}
		else if (MEASURE_READ_STATUS == WAITING2) {
			P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);//flip scan clock SHOULD be done
after waiting1

			MEASURE_READ_STATUS = READING;

			P1OUT |= (1 << MEAS_CLOCK_OUT_PORT);	//set pin output to be lo
			*MEAS_READ_OUT_BUFFER = 0;		//clear readout buffer

			//16 cycles output clock handled in inturrupts
			TA0CCR0 = READOUT_CLK_PERIOD;		// Clock Period
			TA0CTL = TASSEL_2 + MC_1;		// Timer A 0 with ACLK @
1MHz, count UP
			TA0CCTL0 |= BIT4;			// Enable counter interrupts
			}
		else if (MEASURE_READ_STATUS == READING) {
			//Read out as fast as possible - make up for slow system clock
			while(MEASURE_EDGE_COUNT < 32) {
				P1OUT ^= (1 << MEAS_CLOCK_OUT_PORT);		//toggle
measure clock

				*MEAS_READ_OUT_BUFFER = *MEAS_READ_OUT_BUFFER << 1;
				*MEAS_READ_OUT_BUFFER += ((P1IN & (1 << MEAS_IN_PORT)) >>
MEAS_IN_PORT);

				MEASURE_EDGE_COUNT++;
			}

			if(PENDING_MEAS)
				MEASURE_READ_STATUS = INITIATING;
			else
```

```
                                        MEASURE_READ_STATUS = IDLE;
                                P1OUT &= ~(1 << MEAS_CLOCK_OUT_PORT);           //set clock low
                                MEASURE_EDGE_COUNT = 0;
                                TA0CCR0 = READOUT_CLK_PERIOD;
                                TA0CTL = TASSEL_2 + MC_1;                       // Timer A 0 with
ACLK @ 1MHz, count UP
                                TA0CCTL0 |= BIT4;           // Enable counter interrupts

                                //For faster system clock frequencies, use this code
                                /*if(MEASURE_EDGE_COUNT == 32) {
                                        if(PENDING_MEAS)
                                                MEASURE_READ_STATUS = INITIATING;
                                        else
                                                MEASURE_READ_STATUS = IDLE;
                                        P1OUT &= ~(1 << MEAS_CLOCK_OUT_PORT);           //set clock
low
                                        MEASURE_EDGE_COUNT = 0;
                                        TA0CCR0 = SCAN_INIT_WIDTH;
                                        TA0CTL = TASSEL_2 + MC_1;         // Timer A 0 with ACLK @
1MHz, count UP
                                        TA0CCTL0 |= BIT4;           // Enable counter interrupts
                                }
                                else if (MEASURE_EDGE_COUNT % 2) {      //measure on falling edge
                                        P1OUT ^= (1 << MEAS_CLOCK_OUT_PORT);            //toggle clock
                                        *MEAS_READ_OUT_BUFFER = *MEAS_READ_OUT_BUFFER << 1;
                                        *MEAS_READ_OUT_BUFFER += ((P1IN & (1 << MEAS_IN_PORT)) >>
MEAS_IN_PORT);
                                }
                                else {                                          //just toggle clock
on falling edge
                                        P1OUT ^= (1 << MEAS_CLOCK_OUT_PORT);
                                }
                                MEASURE_EDGE_COUNT++;*/
                        }
                }
                else if (MEASURE_SCAN_STATUS == AWAKE && MEASURE_SCAN_INIT == INITIATING) {
                        MEASURE_SCAN_INIT = WAITING1;
                        P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);             //flip scan clock
                }
                else if (MEASURE_SCAN_STATUS == AWAKE && MEASURE_SCAN_INIT == WAITING1) {
                        MEASURE_SCAN_INIT = IDLE;
                        P1OUT ^= (1 << MEAS_SCAN_CLK_PORT);             //flip scan clock
                        P1OUT &= ~(1 << MEAS_SCAN_INIT_PORT);
                }
        }
```

Appendix 3: Test Functions and Example Code

```
#include <msp430g2231.h>
#include "yingzheIC.h"



unsigned int meas_buffer[1];

/*
 * main.c
 */
int main(void) {
 WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
    setup();
       _BIS_SR(GIE);           // LPM0 (low power mode) with interrupts enabled

       unsigned int tx_buffer = 0xAAA9;
       unsigned int rx_buffer1 = 0;
       unsigned int rx_buffer2 = 0;


       const unsigned int comm_gold_code = 0x1234;
       const unsigned int comm_resp      = 0x43A1;
       const unsigned int comm_clear     = 0x0000;
       const unsigned int comm_header    = 0x7777;
       const unsigned int comm_payload1  = 0x5678;
       const unsigned int comm_payload2  = 0xFFFF;

       //demonstrate shifting function
       /*start_comm();
       start_rx_clk(&rx_buffer1);
       tx_buffer = comm_gold_code;
       while(1) {
              start_tx_send(&tx_buffer);
              shift_rx_clk();
       }*/


       //test transmit/receive simultanious
       /*     start_comm();
              start_rx_clk(&rx_buffer1);
              P1DIR += (1 << 0);
              while (1){

                     //clear channel
                     tx_buffer = comm_clear;
                     while(start_tx_send(&tx_buffer));

                     //send gold code
                     tx_buffer = comm_gold_code;
                     while(start_tx_send(&tx_buffer));
```

```
                    while( rx_buffer1 != comm_gold_code ) { //wait to receive gold
code
                          start_tx_send(&tx_buffer);
                          shift_rx_clk();      //perform two shifts to demonstrate
functionality
                    }

                    //clear channel
                    tx_buffer = comm_clear;
                    while(start_tx_send(&tx_buffer));

                    //send response
                    tx_buffer = comm_resp;
                    while( rx_buffer1 != comm_resp ) {      //wait to receive
response
                          start_tx_send(&tx_buffer);
                    }

                    tx_buffer = comm_clear;
                    while(start_tx_send(&tx_buffer));

                    tx_buffer = comm_header;
                    while(start_tx_send(&tx_buffer));
                    while( rx_buffer1 != comm_header) { //wait for header
                          start_tx_send(&tx_buffer);
                    }

                    //fill both buffers once, then clear
                    alternate_rx_rcv_ON(&rx_buffer1, &rx_buffer2, 32);

                    tx_buffer = comm_payload1;
                    while(rx_buffer2 == 0) { //wait for buffer2 to begin filling
                          while(start_tx_send(&tx_buffer));
                    }

                    rx_buffer1 = 0;
                    tx_buffer = comm_payload2;
                    while(rx_buffer1 == 0) {
                          while(start_tx_send(&tx_buffer));
                    }
                    rx_buffer2 = 0;

                    tx_buffer = comm_payload1;
                    while(rx_buffer2 == 0) { //wait for buffer2 to begin filling
                          while(start_tx_send(&tx_buffer));
                    }

                    rx_buffer1 = 0;
                    alternate_rx_rcv_OFF(&rx_buffer1);

                    P1OUT ^= (1 << 0);
              }*/


        //Test measurment
        unsigned int* meas_dest = meas_buffer;
        unsigned int meas_avg;


        wake_measurment();
        init_scan();
        while(1) {
                init_meas(meas_dest);
        }
        sleep_measurment();



        wake_measurment();
        init_scan();
        while(1) {
                meas_dest = meas_buffer;
                meas_avg = 0;
                int i = 0;
                for(i = 0; i < 100; i++) {
                        init_meas(meas_dest);
                        meas_dest++;
                        meas_avg += meas_buffer[i];
                }
                meas_avg /= 100;

        }

        sleep_measurment();

        return 0;
}
```