Data Acquisition System for E80 Rocket Lab

Final Project Report

December 8, 2006

E 155

Andrew Giles and Anu Kohli

Abstract

For the next generation of E80, a new rocket telemetry system needs to be created. This project implements the foundation of a telemetry system. It consists of a magnetometer, a GPS sensor, several timers, and a Blackfin BF537 microprocessor to interface between the sensors. The GPS sensor and magnetometer collect data, which is then stored in memory on the BF537. The GPS interfaces with the processor through a UART whereas the magnetometer interfaces through an SPI port. The timers have been configured, but due to a lack of measurement equipment, their functionality has not yet been completely verified.

Introduction

Harvey Mudd College offers an Experimental Engineering course (E80) which is a core requirement for all engineering majors. This course involves experiments in different disciplines of engineering, which inculcates a deeper understanding of engineering concepts and exposes the student to the practical aspects of engineering. It aids the students in an understanding and application of engineering judgment. One of the labs involved going out on a bridge and determining the vibration shape and damping characteristics of the bridge by obtaining its frequency response. Next year, the bridge lab is being replaced with a rocketry lab. Having taken the course earlier and gained an experience in engineering judgment, field experience and experimental techniques, the team wanted to work on the developing of the lab to help build an experience to help other aspiring engineers on their course to success. In the rocket lab, students will launch a rocket, monitor its behavior using various sensors located on the rocket and then determine the characteristics of the rocket. The team worked on the data acquisition part of the setup. Sensors on the rocket measure various quantities. These measurements need to be saved for future analysis. Most sensors give output in the form of analog signals. This analog data is converted to digital using analog to digital converters (ADC). Data collected from the ADCs is sent to a processor, which then stores the data into a SD memory card, for future analysis.

As part of this project, the team worked on interfacing two data collection devices – the MicroMag3 and the Garmin GPS 15, with a Blackfin BF 537 Processor. The MicroMag3 is a magnetometer that acquires magnetic field data in three, orthogonal directions. It gives out data serially and hence interfaces with the Blackfin Processor via the SPI port. The GPS device acquires GPS data and sends it to the Blackfin over the asynchronous serial port – UART1. The Blackfin processor can be configured to activate and deactivate certain ports using software applications such as Labview Embedded and Visual DSP++. Visual DSP++ was used for this project. The SPI and the UART1 ports were activated for our purposes. Data direction was controlled using control words for the ports and timers were used to activate data collection or data sending depending on the timing requirements of the processing for the master and slave devices. The GPS device required only data collection. No data sending was required to turn on the data acquisition for the GPS device. Hence, data was continuously collected. Currently, data is being saved in an array as an SD card is unavailable. However, eventually, the data collected

will be written to the SD card directly. Also, eventually the SPI port will be interfacing not only with the MicroMag3, but also with two A/D converters, which collect the data from other analog sensors. This will be achieved by sending control words from the processor (master) to each of the SPI devices (slave). Four digital output channels were also initialized, to determine parachute launch, and to generate PWM signals for servos. These were configured using timers.

New Hardware

The Blackfin BF537 processor can be described as a more complicated and faster PIC that includes additional peripheral features. It includes its own programming environment, VisualDSP++, which allows the use of C/C++ code to program the processor. This section will focus on the use of the timers, SPI port, and UART ports on the Blackfin. The memory allocation for variables is abstracted away through the use of the VisualDSP++ IDDE, so it was not necessary to account for that.

Programming the Blackfin is similar to writing C code for the PIC, with several notable differences. To write to a register, a *p must be appended before the register name. Thus, to store 0x061A to the SPI_BAUD register, one would enter

```
*pSPI_BAUD  = 0x061A;
```

Most of the registers used by the Blackfin are 16 bits wide, however, certain configuration registers are 32 bits wide, and it is important to consult the documentation.

In main, the following lines should be included to allow access to all of the registers by name and the common data types:

```
#include <stdlib.h>
#include <sysreg.h>
#include <ccblkfn.h>
#include <BF537 Flags.h>
#include <services\services.h>
```

To setup interrupt handlers and enable interrupts, the SIC_IARx and SIC_IMASK registers need to be written. The SIC_IARx registers control the priority of the interrupts and the SIC_IMASK register defines which interrupts are enabled. The maximum priority a peripheral interrupt can

have is 7, and the lowest priority available is 15. For example, to set the SPI port interrupt to priority 7, the following line of code would be used:

*pSIC_IAR1 = 0xfffff0ff;

This would also disable all of the other interrupts controlled by SIC_IAR0, by setting them to an invalid priority level. The largest defined priority value is 8.

The SIC_IMASK contains 32 bits that control which interrupts are enabled. The hardware manual contains the information on which bits correspond to which interrupt signals.

Once the interrupts are enabled, the interrupt handler routines need to be set up. This is done by setting up a separate .h file containing the following lines:

```
//BF537 Flags.h
#include <sys\exception.h>
#include <cdefBF537.h>
//prototypes
EX_INTERRUPT_HANDLER(SPIgen_ISR);
```

Then, in the same location where the SIC_IARx registers are written, the following lines should be included:

```
#include <BF537 Flags.h>
register_handler(ik_ivg7, SPIgen_ISR);
```

This will allow the creation of a function EX_INTERRUPT_HANDLER(SPIgen_ISR), that is triggered upon the SPI interrupt. ik_ivg7 corresponds to the 7th priority level interrupt. This priority level should be the same as that specified by SIC_IARx, so that the interrupts are mapped to their respective handling routines. Then, the handler routine itself would be defined by:

```
EX_INTERRUPT_HANDLER(SPI_ISR)
{
      data_in = *pSPI_RDBR;
      *pSPI_TDBR = data_out;
}
```

To select between general I/O lines and peripheral functions, the PORTx_FER register must be written. To set a pin for general I/O, the associated PORTx_FER bit should be cleared, while it should be set to enable the peripheral functionality. To control general I/O, there are 6 registers. PORTx_DIR, PORTx_INEN, PORTx_EDGE, PORTx_MASKA, PORTx_MASKB, and PORTxIO. These have the following functions:

- PORTx_DIR controls the direction of the I/O line, whether it is an input, or an output.
- PORTx_INEN enables or disables the input buffer on a pin. This should only be set when the pin is an input to avoid damage to the processor.
- PORTx_MASKA and PORTx_MASKB control which pins are associated with interrupts A and B. These interrupts are then further controlled by the SIC_IARx and SIC IMASK registers.
- PORTx_EDGE controls whether the interrupts are edge or level triggered. This only has an effect when the corresponding PORTx_MASKA and PORTx_MASKB bits are set.
- PORTxIO allows pins to be driven high or low. If it is desired to only set or clear a few pins, PORTxIO_SET and PORTxIO_CLEAR offer the same functionality. These additional two registers are write-to-set PORTxIO and write-to-clear PORTxIO.

Since most of the peripherals on the Blackfin are multiplexed together to the same I/O pins, the PORT_FER register is used to determine which peripheral functions are enabled. This register selects between all of the devices available on the Blackfin, so some care must be taken when deciding which are necessary, as each device or feature prevents another being used.

The SPI port uses several registers for configuration and transfer of data. These are SPI_CTL, SPI_BAUD, SPI_RDBR, and SPI_TDBR. SPI_CTL enables the SPI port, as well as sets various parameters for its operation. SPI_BAUD sets the baud rate, where

Baud rate = SCLK frequency / (2 * SPI_BAUD);

SCLK is 120 MHz by default, but can be modified by editing PLL_DIV.

SPI_RDBR and SPI_TDBR are used to clock data in and out over the associated pins. Depending on the 2 least significant bits of SPI_CTL, the functionality of the interrupts changes. If the two LSB are 00, then the SPI port generates interrupts when SPI_RDBR is full, and triggers data transfers upon a read of RDBR. Thus, to read and write data, the SPI interrupt handler must do the following:

```
EX_INTERRUPT_HANDLER(SPI_ISR)
{
      *pSPI_TDBR = data_out;
      data_in = *pSPI_RDBR;
}
```

The program should write the outgoing data before triggering a data transfer by reading off of SPI_RDBR. Note that the interrupt is not cleared until the SPI_RDBR register is read, so if the interrupt handler does not read the register, the interrupt will reoccur immediately after returning until this does occur.

If the two LSB are 01, the SPI port generates interrupts when SPI_TDBR is empty and initiates data transfers upon a write to SPI_TDBR. Thus, to receive data using this mode, the current data in SPI_ RDBR should be read off, and a dummy word should be written to be SPI_TDBR to clock in the next set of data to receive.

```
EX_INTERRUPT_HANDLER(SPI_ISR)
{
      data_in = *pSPI_RDBR;
      *pSPI_TDBR = data_out;
}
```

Here, if SPI_TDBR is not written to in the interrupt routine, the interrupt will remain set and trigger continuously, resulting in an infinite loop.

In either case, upon configuring SPI_CTL properly, the processor will immediately generate an interrupt. Thus, the interrupt should be able to handle this initial interrupt, even if the exterior serial devices are not prepared to send and receive data at the time.

The UART, or universal asynchronous receiver/transmitter, is somewhat easier to configure. The UART has 3 separate interrupts that are generated upon an error, an empty transmit buffer register, and a full receive buffer register.

There are several registers than need to be written to configure the UART properly. First, UARTx_LCR, the UART control register, needs to be written to enable writes to UARTx_DLL and UARTx_DLH, which control the baud rate of the serial port. The UART is register

equivalent to that found in a PC's serial port, so 4 of the registers share address values and are multiplexed by bit 6 in UARTx_LCR. Here,

Baud rate = SCLK frequency / (16 * UARTx_DLH:UARTx_DLL)

Again, SCLK is 120 MHz by default.

After setting the preferred baud rate, UARTx_LCR should be written again to allow access to UARTx_RBR and UARTx_TBR. Then, to fully enable the UART port, UARTx_GCTL bit 0 should be set high to enable the UART clock.

Once enable, the UART will generate interrupts whenever it has received or transmitted a byte of data. The UART has separate interrupts for receiving and transmitting, allowing them to happen entirely asynchronously from each other. To clear the interrupt, UARTx_RBR must be read (for a receive interrupt) or UARTx_TBR must be written to (for a transmit interrupt).

Example code for the UART:

```
EX_INTERRUPT_HANDLER(UART1_TX_ISR)
{
	data_in = *pUART1_RBR;
}

EX_INTERRUPT_HANDLER(UART1_RX_ISR)
{
	*pUART1_TBR = data_out;
}
```

Configuring the timers is fairly straightforward. All 8 general purpose timers have identical functionality. There are 2 registers that control all 8 timers, and 3 registers associated with each timer. To initialize a timer, the following steps should be taken:

1. Write TIMERx_CONFIG to configure the timer for the desired function.
2. Write TIMERx_WIDTH and TIMERx_PERIOD to set the length of the timer.
3. Write TIMER_ENABLE to enable the desired timer.

When the counter on the timer (viewable by TIMERx_COUNTER) reaches the values specified by TIMERx_WIDTH and TIMERx_PERIOD, the timer will generate an interrupt. This interrupt needs to be set by SIC_IMASK and SIC_IARx, and have an associated interrupt routine. To clear the interrupt, it is necessary to write the TIMER_STATUS register. Thus, to clear the interrupt for TMR0, the code would be

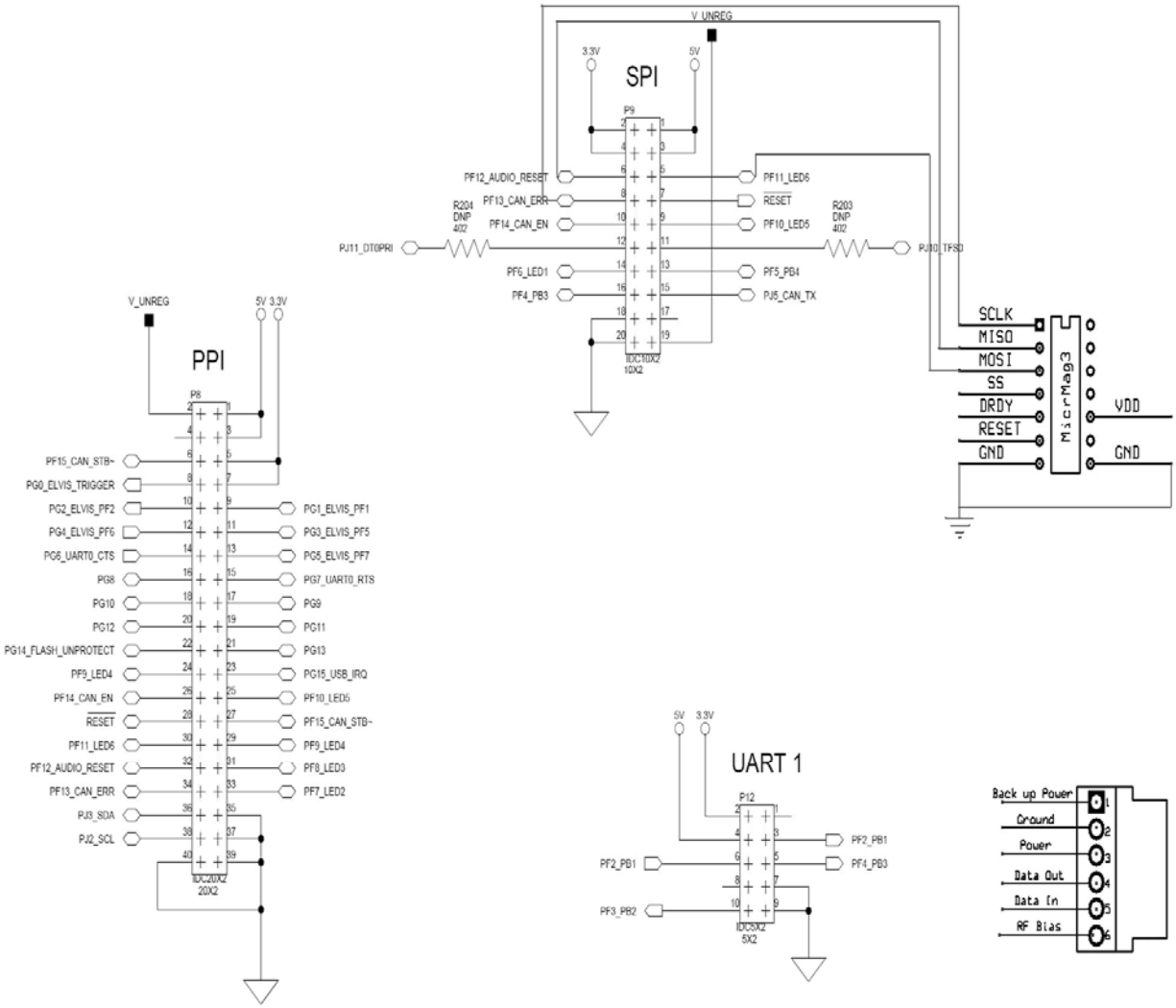
```
EX_INTERRUPT_HANDLER(SPI_ISR)
{
      *pTIMER_STATUS = 0x00000001;
      //rest of code goes here
}
```

The Hardware Reference can be found at the end in the references section [1]. It contains documentation on all of the features of the Blackfin processor, including their functionality and how to configure them. This was the primary resource of the team.

The MicroMag3 [3] is a three axis magnetometer. It measures the magnetic field at its location. It has a settable amount of gain, and can take up to 2000 samples per second at the lowest gain setting. To control the MicroMag3, it is necessary to first reset the sensor by pulsing the reset pin from low to high to low. Then, the control word (8 bytes, see data sheet for more details) is clocked in over an SPI interface to configure the measurement direction and gain. When the measurement is complete, the MicroMag3 sets DRDY high and await the master SPI device to clock the measurement data out.

The Garmin GPS 15 [2] is a small form factor GPS unit. It requires no initial configuration to begin acquiring satellites and determine its position. It sends data out asynchronously at 4800 baud, using a standard set of control ‘sentences.’ To obtain the position data, it is necessary to look for the beginning of the ‘sentence’ that contains the position data, and store that sentence. As the GPS 15 uses ASCII characters to encode the position data, it will be necessary to do some conversion to binary numbers to determine the actual position on a microcontroller. The GPS 15 uses 3.3 V logic levels, so no transceiver is required to communicate with microprocessors over short distances. Reference the data sheet for the ‘sentence’ descriptions and pin-outs for the correct connection.

Schematic



Microcontroller code layout

The code running on the Blackfin is fairly straightforward. There are several sets of timers, the SPI port, and the UART port, all running in parallel and generating interrupts separately. Thus, each can be considered as a standalone unit and discussed by itself.

The SPI port code controls the measurement direction and speed of the magnetometer. To initiate a measurement, first the MicroMag3 must be reset, and a command word sent to it over

the SPI port. After the MicroMag takes the measurement, it raises a pin high to signify that it has completed its measurement, and waits for the data to be clocked out.

To accomplish this, the Blackfin first configures the SPI port, and pulses a general I/O line to reset the MicroMag. Then, the proper control word is sent to the magnetometer to initiate the measurement process, and the Blackfin enters a holding state as it waits for the measurement to finish. After the MicroMag completes its measurement, the ready pin initiates an interrupt on the Blackfin. This interrupt handler proceeds to clock the measurement data out and store it in an array, and begin the process to initiate the next measurement.

To obtain position data from the GPS is somewhat simpler. The GPS automatically configures itself to send out data at 4800 baud, without any input from the host processor. Thus, the GPS data routine simply configures the UART for receiving data at 4800 baud only. Then, whenever a byte of GPS data is received, an interrupt is generated. This interrupt handler stores the GPS data to another array, and waits for the next byte to come in.

The timer interrupts run at a lower priority than the serial ports, and thus do not interfere with their functionality. The timer code configures each of the 4 timers, and then waits for the timer to run out. Then, each of the interrupt handlers either initiates a timer event on a general I/O line, or clears the interrupt so that it can continue its PWM functionality.

Additional timers are utilized for the generation of the four digital output channels. Two of the channels control the launch of two different parachutes on the rocket. This functionality is controlled by timers. Once initialized, the timer interrupt causes a high at the output channel and this will launch the parachute. The other two digital lines are used to generate PWM signals for servos. These PWM signals are again configured using the functionality of the timers.

The main routine of the Blackfin code consists of configuring all of the necessary elements and entering into an infinite loop that waits for the generation of the interrupts. All of the data acquisition work is done by the individual interrupt routines.

Results

The achievement of significant results in this project was a difficult process. Tests were conducted at each level of the development process of the project. However, several problems were encountered. The biggest challenges of this project were to work with a completely new

processor, with new peripherals and new devices of which the team members had little or no experience, and to work in a new software environment. Coding and configuring the SPI and UART ports for data collection was tried using two different software environments – Visual DSP++ and Labview Embedded. Neither interface seemed to work initially, partly because of the lack of good example code provided by the manufacturer. Learning about a new coding environment while understanding the new equipment, caused a dual challenge and delayed the proposed timeline for the project. Given this constraint, majority of the time was spent understanding the interface and trying out different pieces of code and testing it to ensure the functionality of the device and the code. This left the team with little time to actually configure the ports to achieve the desired results. Given a little more time, the team would have reached its set goal. However, the team should have initially selected the deliverables of their project carefully, after examining the vastness of the project. Underestimating the time for problems encountered with the device caused this situation.

In the proposal, the team promised the accomplishment of the following tasks – configure the SPI port to collect data from the MicroMag3 and store to a dummy device, configure the UART and collect data from the Garmin GPS 15 and store the data collected, and configure the four digital output channels for controlling the parachutes and servos. After configuring the ports and the timers, the team accomplished the following tasks, described in detail below. The SPI port was configured for data collection from the MicroMag3 and data was collected and stored into an array. A small problem arose here. Although data was being collected, there was a discrepancy between the rate of data collection. Instead of cycling through the three orthogonal directions of magnetic field, the SPI port occasionally skipped one data point. This is because of the sampling rate or the time interval between data collection. However, in order to ensure that the right data is delivered to the user, the associated direction was recorded with each data point so that the data is not misleading and can be segregated into the different directions accordingly. The changes required to make this work would be to determine the baud rate for the SPI port and the interval between data collection and reconfigure the port accordingly. The UART, which was collecting data from the GPS device worked perfectly. Data was streaming in serially and getting stored into array. Repetition was observed in the data which confirmed our results. Out of the four digital output channels, the two controlling the parachutes were working well. The output went high as soon as the interrupt for the associated timer was called. The other two output

channels were configured for pulse width modulated signals. There is a belief that the timers are properly functional. However, no conformity can be given regarding this as the results could not be tested due to a lack of proper equipment. On testing the output using a logic analyzer, one can determine the perfection of the results. If not functional, the timer will need to be reconfigured to ensure proper functioning. This reconfiguration will be done using the timer registers.

Thus, in conclusion, objectives of the project were achieved, however, not to perfection. Given more time, positive results can be achieved. However, given the time constraints regarding the deadlines, the team cannot change results. But, the above proposed solutions should be taken into consideration when future students take up the project to complete it and fully implement it.

References

[1] Blackfin BF 537 EZ-Lite Kit

(a) ADSP BF537 Blackfin Processor Hardware Reference
<http://www.analog.com/UploadedFiles/Associated_Docs/4206716165649BF537_HRM_whole_book_o.pdf>

{b) ADSP BF537 EZ-Kit Lite Evaluation System Manual
<http://www.analog.com/UploadedFiles/Associated_Docs/303127081ADSP_BF537_EZ_KIT_Lite_Manual_Rev20.pdf>

[2] Garmin GPS 15 <http://www.garmin.com/manuals/GPS15_TechnicalSpecification.pdf>

[3] MicroMag3 <https://www.pnicorp.com/downloadResource/c40c/manuals/110/MicroMag3+3-Axis+Sensor+Module_June+2006.pdf?

Parts List

The Parts List is available with Prof. Erik Spjut in the Engineering Department, with the details of the part number, manufacturer, and price quotes.