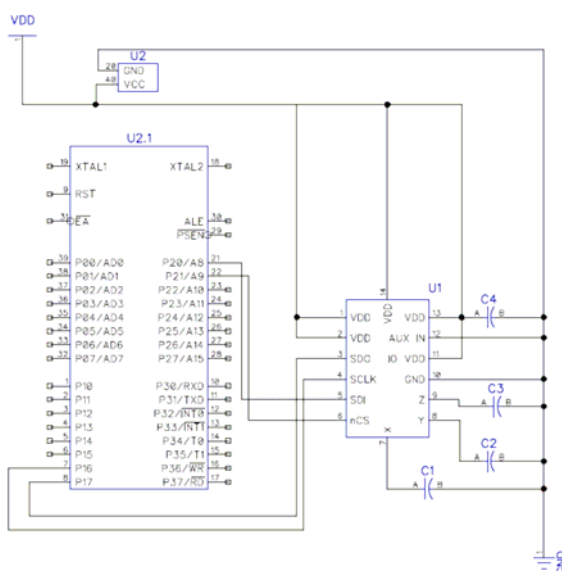


**Introduction**

This application note provides an example interface for a Kionix accelerometer with an example Bluetooth DIP Module. First, this application note talks about the hardware and software interface that is made between a C8051 microcontroller and the Kionix KXPB5 accelerometer. Next, the hardware and software interface between the Bluetooth DIP Module and the microcontroller are discussed. Finally, there is also an attachment of all the AT Commands for the end user to be able to update the code in order to execute numerous functions of the Bluetooth DIP Module. Hardware connections, schematics, timing diagrams and example code are provided in this application note as well. The applications of this interface include serial streaming of acceleration data to a PC for data logging.

**Feature Description**

Communication to the accelerometer sensor is established via SPI communication interface, and the accelerometer always operates as a slave device. SPI is a 4-wire synchronous serial interface that uses two control and two data lines. With respect to the Master (C8051), the Serial Clock output (SCLK), the Data Output (MOSI) and the Data Input (MISO) are shared among the Slave devices. The Master (C8051) generates an independent Chip Select (nCS) for each Slave device that goes low at the start of transmission and goes back high at the end. The Slave Data Output (SDO) line, remains in a high-impedance (hi-z) state when the device is not selected (nCS = high), so it does not interfere with any active devices. This allows multiple Slave devices to share a master SPI port. Please refer to Figure 1 for the schematic.



**Figure 1. KXPB5 communication with a microcontroller**

## Enabling The Accelerometer Sensor

The control register embedded in the accelerometer sensor has an 8-bit address. Upon power up, the Master must write to the accelerometer's control register to set its operational mode. On the falling edge of nCS, a 2-byte command is written to the control register. The first byte, 0x04, initiates the write to the appropriate register, and is followed by the user-defined, operational-mode byte, 0x04, which enables the accelerometer sensor. All commands are sent MSB (most significant bit) first, and the host must return nCS high for at least 200nS before the next data request.

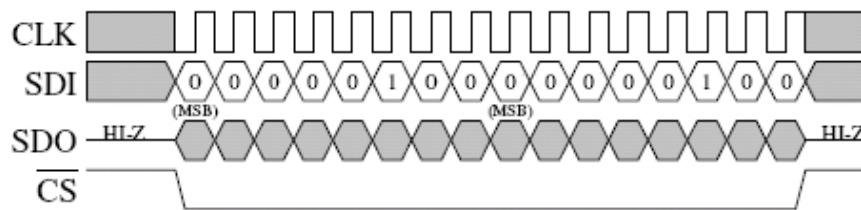


Fig. 2) Timing Diagram for 8-bit Control Register Write Operation

In order to read the 8-bit control register, an 8-bit read command, 0x03, must be written to the accelerometer to initiate the read. Upon receiving the command, the accelerometer returns the 8-bit operational-mode data stored in the control register. This operation also occurs over 16 clock cycles. All returned data is sent MSB first, and the host must return nCS high for at least 200 nS before the next data request. Figure 3 show the timing diagram for an 8-bit control register read operation.

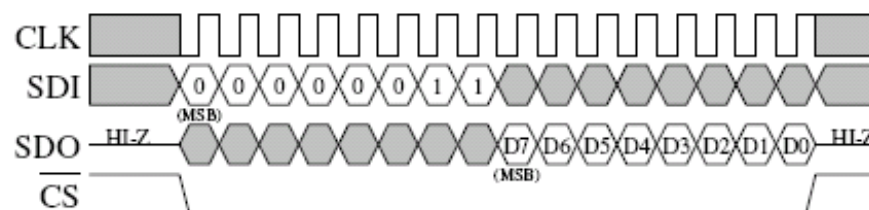


Fig. 3) Timing Diagram for 8-bit Control Register Read Operation

## Reading Acceleration Data

In order to read the 3-axis (X, Y, and Z) acceleration data, transmission of an 8-bit axis conversion command (see Table 1) must be issued on the falling edge of nCS. After the eight clock cycles used to send the command, the host must wait for at least 40us in order to clock in the acceleration data. Note that all returned data is sent MSB first. Once the data is

received, nCS must be returned high for 200ns before the next data request. Figure 3 shows the timing for the accelerometer read operation. Please also note that the acceleration data has 12-bit resolution (12-bits long).

The Read Back Operation is a 3-byte SPI command. The first byte contains the command to convert one of the axes. The second and third bytes contain the 12 bits of acceleration data plus four least significant bits of padding to make a total of 16 bits.

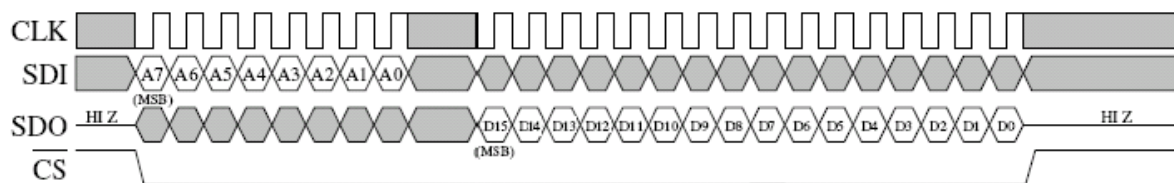


Fig. 4) Timing diagram for 16-bit data read operation

Once the 16-bits of data have been clocked in, they are shifted left four times, since the least four significant bits are garbage.

Description	1 <sup>st</sup> byte (Command)
Convert X axis	0x00
Convert Z axis	0x01
Convert Y axis	0x02
Read Control Register	0x03
Write Control Register	0x04
Convert Aux In	0x07

Table 1 Command Register Bit Utilization

## Connections

There are three pieces of hardware and two pieces of software you need in order to develop this wireless module. The three pieces of hardware you need are the Kionix KXPB5, the SiLabs C8051F312, a Mitsumi Bluetooth Dip Module, and the Bluetooth USB Module. The two pieces of software that are needed are the BlueSoleil (provided with the USB Module) and the software/firmware that is provided at the end of this app note. Instructions on installing the Bluetooth USB Module are described in the following link:

<http://www.sparkfun.com/commerce/present.php?p=Software>.

Figure 5 shows the block diagram of the hardware connections between the microcontroller and the Bluetooth DIP Module.

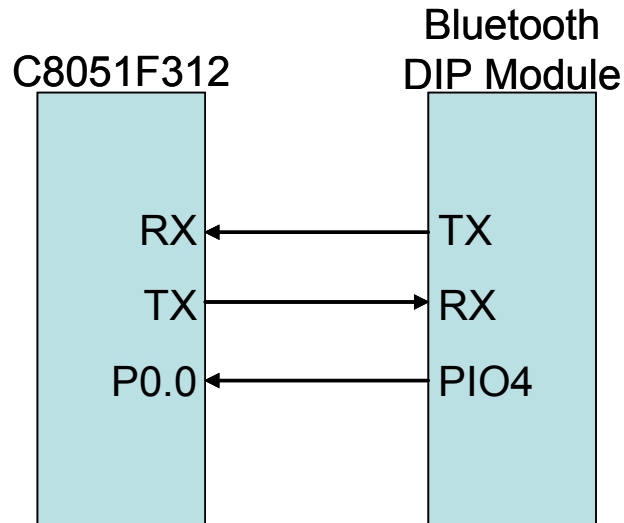


Fig. 5) Block Diagram

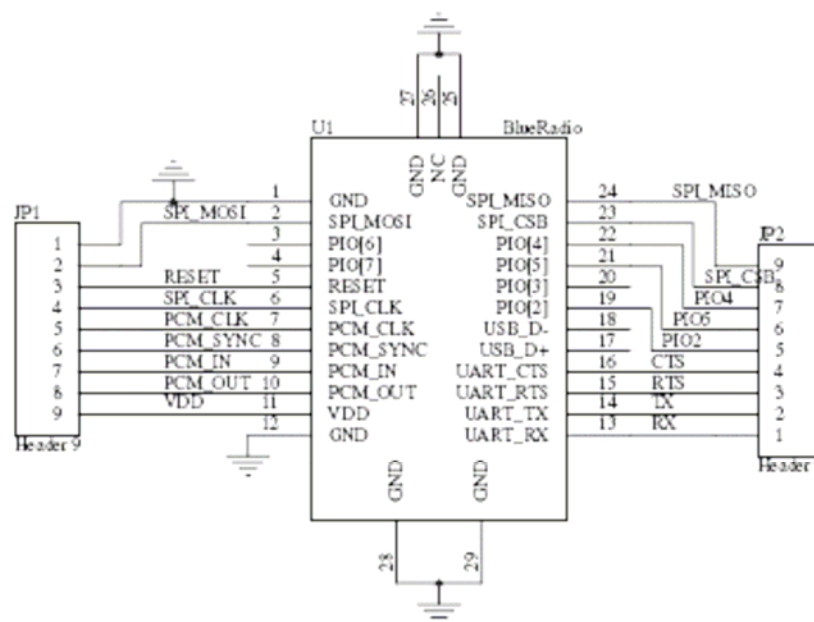


Fig. 6) Bluetooth Dip Schematic

In order to establish communication between the KXPB5 and C8051F312 and the Bluetooth Dip Module, the following hardware connections should be made as shown in Figure 7. Power and logic signals used for powering and communication between the KXPB5, the C8051F312 and the Bluetooth Dip Module are 3.3V. The AT commands necessary to put the Bluetooth Dip Module into data mode are handled inside the firmware. Communication between the Bluetooth DIP Module and the Bluetooth USB Module is done via the discovery and connection procedure outlined in the link given previously for installing the Bluetooth USB Module. Once the connection has been established, a user may open the HyperTerminal and configure it to the appropriate COM port at 9600/8/none/1 to view the accelerometer X, Y and Z outputs being sent by the Bluetooth DIP Module.

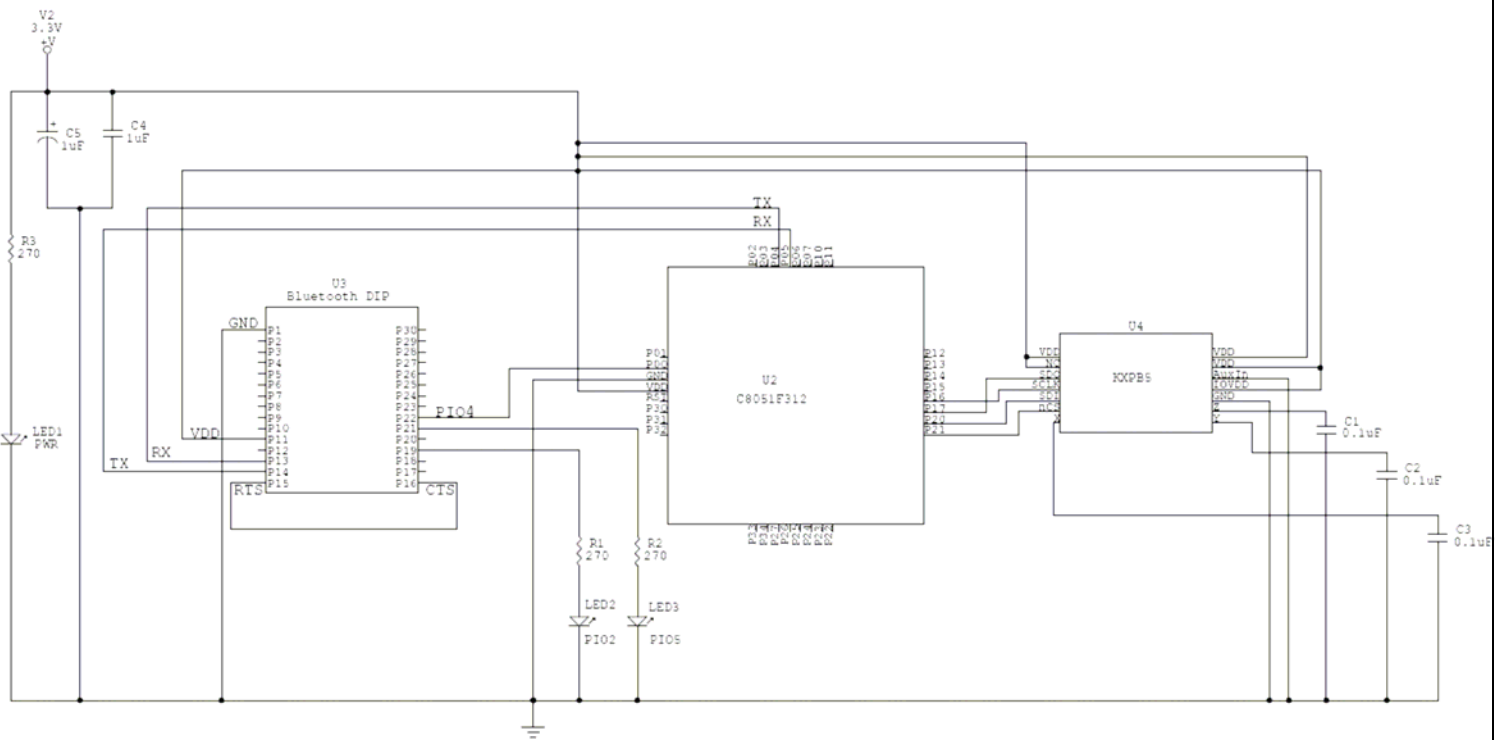


Fig. 7) Bluetooth Wireless Module Schematic

## Software Operation

The four software routines used to access the Bluetooth DIP Module are 'Init\_Device', 'Enable\_accelerometer', 'Bluetooth\_connected' and 'Read\_axis\_data'. The 'Init\_Device' routine initializes the C8051 device interface logic and port configurations. This routine is only called in the initialization sequence (power up) of the device. The 'Enable\_accelerometer' routine sets the enable bit in the control register of the accelerometer sensor to place the sensor in operational/running mode.

```
CS = 0;           //Select the slave device(accelerometer)
SPI_Transfer(0x04); //command to write to KXPB5 control register
SPI_Transfer(0x04); //enable
CS = 1;           //communication complete
```

The 'Bluetooth\_connected' routine waits for the user to discover and connect to the Bluetooth DIP Module using the BlueSoleil software. Once the device DIP Module has been discovered and connected to, the PI04 pin on the DIP Module will be pulled high and will notify the firmware to place it into data mode. For example:

```
while(!PO4);           //wait for the connection to be established with
                       //the Bluetooth DIP Module. Otherwise do nothing.

for(a=0; a<8333; a++){ //wait one second before putting the
    wait(200);          //Bluetooth DIP Module into data mode
}

printf("ATMD\r\n");    //put radio into data mode
```

The 'Read\_axis\_data' routine communicates to the accelerometer sensor via SPI obtaining the 12 bits of X, Y, and Z acceleration data. This data is then communicated to the Bluetooth DIP Module via UART, which is then sent to the PC.

```
//-----  
// Bluetooth.c  
//-----  
//  
// http://www.kionix.com  
//  
// Program Description:  
//  
// Example software to demonstrate the interface between a KXPB5 and C8051F312  
// and an example Bluetooth DIP Module.  
// - Interrupt-driven SPI implementation  
// - Only master states defined  
// - Timer2 used as data conversion clock source  
// - Timer1 used by UART data rate transfer  
//  
// - Pinout:  
// P1.6 -> SCLK (SPI)  
// P1.7 -> SDO (SPI)  
// P2.0 -> SDI (SPI)  
// P2.1 -> nCS (SPI)  
//  
// P0.0 -> PO4 (Bluetooth connection status)  
// P0.4 -> TX (UART)  
// P0.5 -> RX (UART)  
//  
// all other port pins unused  
//  
// How To Test:  
//  
// 1) Download code to a KXPB5 and C8051F312 device that is connected to a  
// Bluetooth Dip Module.  
// 2) Run the code:  
// a) Establish communication between the Bluetooth USB Module  
// and the Bluetooth DIP Module.  
// b) Open the HyperTerminal and configure it  
// to the proper COM port at 9600/8/None/1/None to view the  
// X, Y, and Z acceleration output in counts (12 bit resolution)  
// Therefore for a +/-2g part  $g = ((\text{output in counts} - 2048)/819)$   
//  
//  
// Target: C8051F31x  
// Tool chain: Keil C51 7.50 / Keil EVAL C51  
// Command Line: None  
//
```

```

// Release 1.0
// -Initial Revision (Rev 0.0)
// -10 MAY 2007
//

//-----
// Includes
//-----
#include "C8051F310.h"
#include <stdio.h>
#include <math.h>
//-----
// Global VARIABLES
//-----
sbit          CS    = P2^1;
sbit  PO4 = P0^0;
sbit  RTS = P1^0;
sbit  CTS = P0^7;

unsigned int output;
unsigned int Xout, Yout, Zout;
long a, b;
//-----
// Function PROTOTYPES
//-----

unsigned int SPI_Transfer(int txbyte);
void wait(int counts);
unsigned int Xaxis();
unsigned int Yaxis();
unsigned int Zaxis();

//-----
// 16-bit SFR Definitions for 'F31x
//-----

sfr16      DP          = 0x82;  // data pointer
sfr16      TMR2RL      = 0xca;  // Timer2 reload value
sfr16      TMR2        = 0xcc;  // Timer2 counter
sfr16      TMR3        = 0x94;  // Timer3 counter
sfr16      TMR3RL      = 0x92;  // Timer3 reload value
sfr16      PCA0CP0     = 0xfb;  // PCA0 Module 0 Capture/Compare
sfr16      PCA0CP1     = 0xe9;  // PCA0 Module 1 Capture/Compare
sfr16      PCA0CP2     = 0xeb;  // PCA0 Module 2 Capture/Compare
sfr16      PCA0CP3     = 0xed;  // PCA0 Module 3 Capture/Compare
sfr16      PCA0CP4     = 0xfd;  // PCA0 Module 4 Capture/Compare

```



```

sfr16      PCA0          = 0xf9; // PCA0 counter
sfr16      ADC0          = 0xbd; // ADC Data Word Register
sfr16      ADC0GT       = 0xc3; // ADC0 Greater-Than
sfr16      ADC0LT       = 0xc5; // ADC0 Less-Than

//-----
// Peripheral specific initialization functions,
// Called from the Init_Device() function
//-----

void Timer_Init()
{
    //Configure the UART for 9600 baud data rate transfer
    TMOD    = 0x20;           //configure timer1 as an 8-bit counter/timer
                               //with auto-reload
    TH1     = 0x96;           //Load the high byte of timer1 with 96
}

void UART_Init()
{
    SCON0   = 0x10;           //Enable UART reception
}

void SPI_Init()
{
    SPI0CFG = 0x40;           //configure SPI as a master
    SPI0CN  = 0x01;           //enable SPI
    SPI0CKR = 0x02;           //configure SPI clock to 4 MHz
}

void Port_IO_Init()
{
    // P0.0 - Skipped,   Open-Drain, Digital
    // P0.1 - Skipped,   Open-Drain, Digital
    // P0.2 - Skipped,   Open-Drain, Digital
    // P0.3 - Skipped,   Open-Drain, Digital
    // P0.4 - TX0 (UART0), Push-Pull, Digital
    // P0.5 - RX0 (UART0), Open-Drain, Digital
    // P0.6 - Skipped,   Open-Drain, Digital
    // P0.7 - Skipped,   Open-Drain, Digital

    // P1.0 - Skipped,   Open-Drain, Digital
    // P1.1 - Skipped,   Open-Drain, Digital
    // P1.2 - Skipped,   Open-Drain, Digital
    // P1.3 - Skipped,   Open-Drain, Digital
    // P1.4 - Skipped,   Open-Drain, Digital
    // P1.5 - Skipped,   Open-Drain, Digital
}

```

```
// P1.6 - SCK (SPI0), Push-Pull, Digital
// P1.7 - MISO (SPI0), Open-Drain, Digital
// P2.0 - MOSI (SPI0), Push-Pull, Digital
// P2.1 - Unassigned, Push-Pull, Digital
// P2.2 - Unassigned, Open-Drain, Digital
// P2.3 - Unassigned, Open-Drain, Digital

P0MDOUT = 0x10;
P1MDOUT = 0x40;
P2MDOUT = 0x03;
P0SKIP  = 0xCF;
P1SKIP  = 0x3F;
XBR0    = 0x03;
XBR1    = 0x40;
}

void Oscillator_Init()
{
    OSCICN = 0x83;    //configure the internal clock for 24.5 MHz
}

void Interrupts_Init()
{
    IE     = 0xD0;    //enable UART and SPI interrupt
}

// Initialization function for device,
// Call Init_Device() from your main program
void Init_Device(void)
{
    Timer_Init();
    UART_Init();
    SPI_Init();
    Port_IO_Init();
    Oscillator_Init();
    Interrupts_Init();
}

//-----
// MAIN Routine
//-----
//
// Main routine performs all configuration tasks, then loops forever receiving
// acceleration data and forwarding it to the Bluetooth DIP Module.
//
```

```

void main (void)
{
PCA0MD &= ~0x40;           // WDTE = 0 (dissable watchdog timer)
Init_Device ();           // Initialize the peripherals
  TR1 = 1;                 // START Timer1
  TI0 = 1;                 // Indicate TX0 ready

CS = 0;
SPI_Transfer(0x04);       //command to write to KXPB5 control register
SPI_Transfer(0x04);       //enable
CS = 1;

for(a=0; a<8333; a++){           //wait one second for the pins to stabilize
wait(200);
}

while(!PO4);                   //wait for the connection to be established with
                               //the Bluetooth DIP Module. Otherwise do
nothing.

for(a=0; a<8333; a++){           //wait one second before putting the
wait(200);                       //Bluetooth DIP Module into data mode
}

printf("ATMD\r\n");             //put radio into data mode

while (1){

Xout = Xaxis();                //read x axis
Yout = Yaxis();                //read y axis
Zout = Zaxis();//read z axis

// send x, y, and z axis data to the bluetooth dip module
// all of the AT commands and data that are sent to the
// Bluetooth DIP Module need a carriage return '\r' and
// a new line feed '\n'.
printf("X = %d, Y = %d, Z = %d\r\n", Xout, Yout, Zout);

    }
}

unsigned int SPI_Transfer(int txbyte)
{
SPIIF = 0;                     // clear flag
SPI0DAT = txbyte;              // transmit a byte

```

```
while (!SPIF);           // wait until end of transmission
return SPI0DAT;         // retrieve data from the slave
}

void wait(int counts)
{
    TF2H = 0;
    TMR2CN = 0x00;
    TMR2 = -counts;     // timer will overflow in counts timer clock cycles
    (SYSCLK/12)
    TR2 = 1;           // turn on timer
    while(!TF2H);     // wait until timer interrupts
    TR2 = 0;           // turn timer off
}

unsigned int Xaxis()
{
    CS = 0;
    SPI_Transfer(0x00); //command to convert X-Axis
    // wait some cycles
    wait(200);
    output = SPI_Transfer(0x00); // read first 8 bits
    output = 0x100*output + SPI_Transfer(0x00); // read next 8 bits
    output >>= 4; // shift the output from 16 to 12 bits (4 orders lower)
    CS = 1;
    return output;
}

unsigned int Yaxis()
{
    CS = 0;
    SPI_Transfer(0x02); //command to convert X-Axis
    // wait some cycles
    wait(200);
    output = SPI_Transfer(0x00); // read first 8 bits
    output = 0x100*output + SPI_Transfer(0x00); // read next 8 bits
    output >>= 4; // shift the output from 16 to 12 bits (4 orders lower)
    CS = 1;
    return output;
}

unsigned int Zaxis()
{
    CS = 0;
    SPI_Transfer(0x01); //command to convert X-Axis
    // wait some cycles
    wait(200);
```

```
output = SPI_Transfer(0x00);           // read first 8 bits
output = 0x100*output + SPI_Transfer(0x00); // read next 8 bits
output >>= 4;                          // shift the output from 16 to 12 bits (4 orders lower)
CS = 1;
return output;
}

//-----
// END OF FILE
//-----
```

## AT COMMAND SUMMARY TABLE

AT Command	Description	Requires Reset	Stores Permanently
<b>Attention Prefix</b>			
AT	Attention Prefix	N/A	N/A
<b>Firmware Version</b>			
ATVER,ver1	Module Firmware Version	N/A	Yes
<b>Resetting</b>			
ATURST	Unit Reset	N/A	N/A
ATFRST	Factory Reset	N/A	N/A
ATSSW,0	Set Bypass PIO(4) Factory Reconfiguration	Yes	Yes
ATRSW,0	Read Bypass PIO(4) Factory Reconfiguration	N/A	Yes
<b>Boot Mode</b>			
ATSSW,1	Set Boot Mode	Yes	Yes
ATRSW,1	Get Boot Mode	N/A	N/A
<b>Security Level</b>			
ATSSW,2	Set Security Level	Yes	Yes
ATRSW,2	Get Security Level	N/A	N/A
<b>Get Status</b>			
ATSI,0	Get Module Type	N/A	Yes
ATSI,1	Get Bluetooth Address	N/A	Yes
ATSI,2	Get Friendly Name	N/A	Yes
ATSI,3	Get Current Status of Connections	N/A	Yes
ATSI,4	Get Service Name	N/A	Yes
ATSI,5	Get Class of Device (COD)	N/A	Yes
ATSI,6	Get Response, Security, Auto SCO, Filter Settings	N/A	Yes
ATSI,7	Get Connection, Comm, UART, Service Modes	N/A	Yes
ATSI,8	Get UART Settings	N/A	Yes
ATSI,9	Get Master Auto-Connect Address	N/A	Yes
ATSI,10	Get Slave Scan Intervals and Windows	N/A	Yes
ATSI,11	Get PIO(5) Pulse Rate	N/A	Yes
ATSI,12	Get Escape Character	N/A	Yes
ATSI,13	Get Timeout Settings	N/A	Yes
ATSI,14	Get Maximum TX Power Level	N/A	Yes
ATSI,15	Get PIN Lock Mode	N/A	Yes
ATSI,16	Get Deep Sleep Mode	N/A	Yes
ATSI,17	Get Sniff Settings	N/A	Yes
ATSI,18	Get Link Supervisory Timeout	N/A	Yes
ATSI,19	Get List of Paired or Secured Addresses	N/A	Yes
<b># of Connections</b>			
ATSSW,3	Set Max Connection Number	Yes	Yes
ATRSW,3	Read Max Connection Number	N/A	Yes
<b>Radio Name</b>			

ATSN	Set Radio Name	No	Yes
ATRRN	Read Remote Radio Name By BT Address	N/A	Yes
<b>Service Name</b>			
ATSSN	Set Service Name	Yes	Yes
ATSSNC	Set Service Name by Channel	Yes	Yes
ATRSN	Read Service Name	N/A	Yes
ATRSNC	Read Service Name by Channel	N/A	Yes
ATRRSN	Read Remote Service Name	N/A	Yes
<b>Security</b>			
ATSP	Set PIN	No	Yes
ATOP	Overwrite PIN	No	Yes
<b>COD</b>			
ATSC	Set Class of Device (COD)	Yes	Yes
<b>Write Memory</b>			
ATSW,20	Switch 20: Write UART Settings	No	Optional
ATSW,21	Switch 21: Write Slave Scan Intervals & Windows	Yes	Yes
ATSW,22	Switch 22: Write PIO State	No	Optional
ATSW,23	Switch 23: Write PIO Level	No	Optional
ATSW,24	Switch 24: Write Default Settings	For Security	Yes
ATSW,25	Switch 25: Write Power Up Default Modes	Yes	Yes
ATSW,26	Switch 26: Lock User Settings	No	Yes
ATSW,27	Switch 27: Write LED Rate	No	Yes
ATSW,28	Switch 28: Write Inquiry Timeout Settings	No	Yes
ATSW,29	Switch 29: Write PIN Lock Mode	No	Yes
ATSW,30	Switch 30: Write Deep Sleep Mode	No	Yes
<b>Read Memory</b>			
ATSR21	Read PIO Level	N/A	N/A
<b>Inquiry</b>			
ATDI	Dial Inquiry	N/A	N/A
ATIL	Last Inquiry	N/A	Yes
<b>Master Connect</b>			
ATDM	Dial As Master	N/A	N/A
ATDC	Dial Channel	N/A	N/A
ATDL	Dial Last	N/A	Yes
ATLAST	Read Last Connected Address	N/A	Yes
<b>Master Default</b>			
ATSMA	Set Master Default Address	Yes	Yes
ATMACLR	Master Address Clear	No	Yes
<b>Connect Slave</b>			
ATDS	Dial As Slave	N/A	N/A
<b>Disconnect</b>			

ATDH	Dial Hang Up	N/A	N/A
ATDHC	Dial Hang Up By Channel	N/A	N/A
<b>Modes</b>			
+++	Default Escape Character	N/A	N/A
ATSESC	Set Command Mode Escape Character	No	Yes
ATMD	Put Radio Into Data Mode	No	No
ATMF	Put Radio Into Fast Data Mode	No	No
<b>Cancel</b>			
ATUCL	Cancel (Idle Mode)	No	No
<b>Pairing</b>			
ATPAIR	Pair Radios	No	Yes
ATUPAIR	Unpair By Index	No	Yes
ATUPAIRB	Unpair By Bluetooth Address	No	Yes
ATCPAIR	Clear all paired or secured connections	No	Yes
<b>Sniff and Park</b>			
ATSNIFF	Enable Sniff	No	Yes
ATSSNIFF	Enable Auto Sniff	No	Yes
ATCSNIFF	Clear Sniff	No	Yes
ATXSNIFF	Exit Sniff	No	N/A
ATPARK	Park	No	No
ATXPARK	Exit Park	No	N/A
<b>RSSI and Link</b>			
ATRSSI	Get RSSI Value	N/A	No
ATRSSIC	Get RSSI Value by Channel	N/A	No
ATLQ	Get Link Quality	N/A	No
ATLQC	Get Link Quality by Channel	N/A	No
<b>Audio PCM</b>			
ATDSCO	Dial SCO	N/A	N/A
ATDHSCO	Dial Hang Up SCO	N/A	N/A
<b>Max TX Power</b>			
ATSPF	Set Max TX Power Level	No	Yes
<b>Link Timeout</b>			
ATLSTO	Link Supervisory Timeout	No	Yes
<b>Variable Storage</b>			
ATSTORE	Store Variable	No	Yes
ATREAD	Read Variable	N/A	Yes



## Conclusions

The measurements obtained in this study verified two linear accelerometers can be used to determine angular rotation rates. This method works best when the rotational motions are quick with large angular accelerations. In this case, there is no chance of dividing by zero. Integration only takes place for short time, so drift is not a problem. Angular accelerations need to be lower than the accelerometer sensing range.

## The Kionix Advantage

Kionix technology provides for X, Y, and Z-axis sensing on a single, silicon chip. One accelerometer can be used to enable a variety of simultaneous features including, but not limited to:

- Hard Disk Drive protection
- Vibration analysis
- Tilt screen navigation
- Sports modeling
- Theft, man-down, accident alarm
- Image stability, screen orientation & scrolling
- Computer pointer
- Navigation, mapping
- Game playing
- Automatic sleep mode

## Theory of Operation

Kionix MEMS linear tri-axis accelerometers function on the principle of differential capacitance. Acceleration causes displacement of a silicon structure resulting in a change in capacitance. A signal-conditioning CMOS technology ASIC detects and transforms changes in capacitance into an analog output voltage, which is proportional to acceleration. These outputs can then be sent to a micro-controller for integration into various applications. For product summaries, specifications, and schematics, please refer to the Kionix MEMS accelerometer product sheets at <http://www.kionix.com/sensors/accelerometer-products.html>.