PANDORA PRODUCTS

Zigbee Notes
Zigbee Use

Jim Schimpf

Document Number: PAN-201712001
Revision Number: 0.5
9 March 2018

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724-539.1276

Email: jim.schimpf@gmail.com

Document Revision History

Version	Author	Description	Date
0.1	js	Initial Version	21-Dec-2017
0.2	js	GettingStarted testing	12-Jan-2018
0.3	js	Add software description	24-Feb-2018
0.4	js	Add software internals description	25-Feb-2018
0.5	js	CONFIG & STATUS registers	9-Mar-2018

Contents

List of Figures

1 Introduction

Zigbee is a radio protocol that uses direct squence spread spectrum transmission and a mesh network to transfer data to/from IoT devices(http://www.zigbee.org).[1]The radio can operate in three bands, 2.5 GHz (worldwide), 902-928 (USA), and 868 (Europe).

2 Focus of Paper

The rest of this paper will concentrate on Zigbee use with the Nordic nRF24L01 tranceiver chip.[2]This was available on Amazon as a breakout board (https://www.amazon.com/Makerfire-Arduino-NRF24L01-dp/B00O90868G/ref=sr_1_1?ie=UTF8&qid=1513673355&sr=8-1&keywords=zigbee+module. With 10 of them available for ~$12. This unit looks like this:

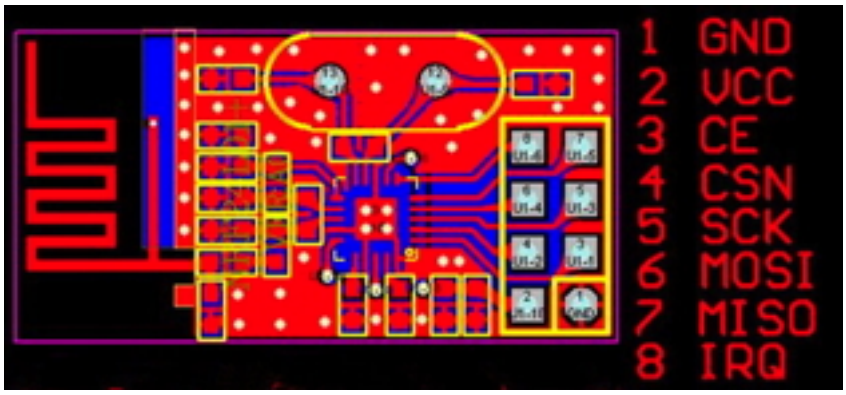

Figure 1: nRF24L01 Module

2.1 Initial Test

2.1.1 Wiring

The units run on 3V but have 5V tolerant inputs. Thus can be run by Arduinos. The first test uses the RF24 Arduinio Library with the program "GettingStarted". The Arduino hookup is:

Signal	Module Pin	Arduino Pin
GND	1	GND
VCC	2	3.3V
CE	3	7
CAN	4	8
SCK	5	13
MOSI	6	11
MISO	7	12

Table 1: Arduino Hookup

2.1.2 Software

The software used is "GettingStarted" in the RF24 library examples. A copy is made of the Arduino application and the original and the new copy are attached to the two Arduino's hooked up to the Zigbee radios. One of the copies is modified by changing the variable *radioNumber* from 0 to 1. Then both are started. One unit will transmit a random uint32 to the other and it will respond by sending the value back.

```
Sent 48899116, Got response 48899116, Round-trip delay 33224 microseconds
Sent response 48899116
Sent 49934396, Got response 49934396, Round-trip delay 33188 microseconds
Sent response 49934396
Sent 50969640, Got response 50969640, Round-trip delay 1800 microseconds
Sent response 50969640
Sent 51973440, Got response 51973440, Round-trip delay 3784 microseconds
Sent response 51973440
:
:
```

3 LPC1114 Version

3.1 SPI/SSP

3.1.1 Processor Pins

The first item that must be done is to set up the pins on the processor for SPI0 with the zigbee & EEPROM use. The following pins are used.

LPC1114 Pin	Name	Use	Zigbee Board
1	PIO0_8	MISO0	MI -7
2	PIO0_9	MOSI0	MO-6
6	PIO_6	SCK0	SCK-5
26	PIO0_3	GPIO	CE-3
28	PIO0_7	GPIO	CSN-4

Table 2: SPI pins

The SCK pin is a secondary choice because the other SCK0 is pin 3 which is already used for SWCLK a debug line. Set up these lines in the processor multiplexer section with following code.

```
#define LPC_SSP           LPC_SSP0
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_8, (IOCON_FUNC1 | IOCON_MODE_INACT));/* MISO0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_9, (IOCON_FUNC1 | IOCON_MODE_INACT));/* MOSI0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_2, (IOCON_FUNC1 | IOCON_MODE_INACT));/* SSEL0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_6, (IOCON_FUNC2 | IOCON_MODE_INACT));/* SCK0 */
Chip_IOCON_PinLocSel(LPC_IOCON, IOCON_SCKLOC_PIO0_6);
Chip_SSP_Init(LPC_SSP);
```

Note that PIO0_6 needs to be set with IOCON_FUNC2 to enable it. All the lines are set MODE_INACT since they are under the control of the SPI block not the GPIO.

3.1.2 SPI Setup

Next the SPI must be set up for the clock phase, frame format and data bits. The following code does that.

```
// These are in the ssp_11xx.h file
typedef enum CHIP_SSP_FRAME_FORMAT {
SSP_FRAMEFORMAT_SPI = (0 << 4),
/**< Frame format: SPI */
CHIP_SSP_FRAME_FORMAT_TI = (1u << 4),/**< Frame format: TI SSI */
SSP_FRAMEFORMAT_MICROWIRE = (2u << 4),/**< Frame format: Microwire */
} CHIP_SSP_FRAME_FORMAT_T;
typedef enum CHIP_SSP_CLOCK_FORMAT {
SSP_CLOCK_CPHA0_CPOL0 = (0 << 6),/**< CPHA = 0, CPOL = 0 */
SSP_CLOCK_CPHA0_CPOL1 = (1u << 6),/**< CPHA = 0, CPOL = 1 */
SSP_CLOCK_CPHA1_CPOL0 = (2u << 6),/**< CPHA = 1, CPOL = 0 */
SSP_CLOCK_CPHA1_CPOL1 = (3u << 6),/**< CPHA = 1, CPOL = 1 */
SSP_CLOCK_MODE0 = SSP_CLOCK_CPHA0_CPOL0,/**< alias */
SSP_CLOCK_MODE1 = SSP_CLOCK_CPHA1_CPOL0,/**< alias */
SSP_CLOCK_MODE2 = SSP_CLOCK_CPHA0_CPOL1,/**< alias */
SSP_CLOCK_MODE3 = SSP_CLOCK_CPHA1_CPOL1,/**< alias */
} CHIP_SSP_CLOCK_MODE_T;
```

```
// ********************************
#define SSP_MODE_TEST       1 /*1: Master, 0: Slave */
SSP_ConfigFormat ssp_format;
ssp_format.frameFormat = SSP_FRAMEFORMAT_SPI;
ssp_format.bits = SSP_DATA_BITS;
ssp_format.clockMode = SSP_CLOCK_MODE0;
Chip_SSP_SetFormat(LPC_SSP, ssp_format.bits,
ssp_format.frameFormat,
ssp_format.clockMode);
Chip_SSP_SetMaster(LPC_SSP, SSP_MODE_TEST);
Chip_SSP_Enable(LPC_SSP);
Chip_SSP_DisableLoopBack(LPC_SSP);
```

Last the clock speed is set with a call to Chi0_SSP_SetClockRate(); The call has the following parameters:

```
/**
* @brief Set up output clocks per bit for SSP bus
* @param pSSP : The base of SSP peripheral on the chip
* @param clk_rate fs: The number of prescaler-output clocks per bit on the bus, minus one
* @param prescale : The factor by which the Prescaler divides the SSP peripheral clock PCLK
* @return Nothing
* @note The bit frequency is PCLK / (prescale x[clk_rate+1])
*/
```

The output clocks for a number of input values are.

clk_rate	prescale	SCK mHz
1	2	11.99
1	4	6.099
1	8	3.012
1	16	1.507
1	32	754.0 (kHz)
2	2	8.054
2	4	4.032
2	8	2.012
2	16	1.007
2	32	502.1 (kHz)
3	2	6.030
3	4	3.017
3	8	1.509
3	16	753.5 (kHz)
3	32	376.8 (kHz)

Table 3: Clock speeds

LPC1114 Pin	Use	EEPROM Pin
	A0-NC	1
	A1-NC	2
	A2-NC	3
	GND	4
	VCC	8
	GND	7
27	SCL-PIO0_4	6
5	SDA-PIIO0_5	5

Table 4: EEPROM Pins

This is with the default 48 mHz clock speed and using the built in oscillator.

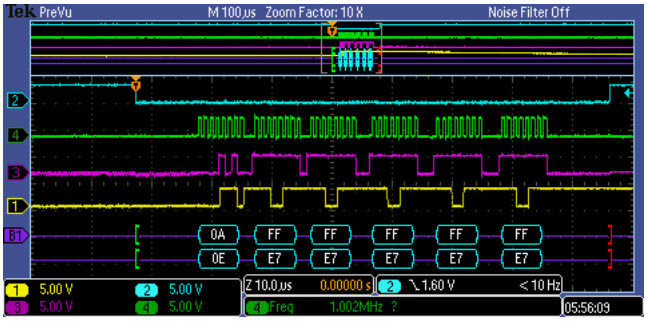

Figure 2: SPI I/O to nRF24

NOTE: Before any use of the nRF24 you must write 0x0C to the CONFIG (0x00) register to move the chip out of sleep mode. Any I/O before doing this will return bogus values.

This shows an example I/O (read of the TX_ADDR register) The first byte is the read command [2]and the following bytes are slots for the 5 TX addresses. Note that the blue line (CSN) is low for the entire transaction. The byte returned(yellow line/lower decode data) on the first byte is 0x0E the status byte. The I/O operation works up to 20 MHz so that the 8MHz setting of the SPI works also, but here a slower speed 1.002 MHz is chosen for a nicer display of the data.

3.1.3 EEPROM

3.1.4 Processor Pins

3.1.4.1 Introduction The board needs memory that will not be lost on a power cyle. This is necessary to preserve values when the battery is changed. EEPROM can do this and allows byte level memory changes. It was found that the 24C16 (2K byte) memory was available, in an 8 pin package and at a very inexpensive price so it was chosen.

The interface is I2C which is already used for the MMA8451 accelerometer and is detailed in [**?**]. The 24C16 chip only has the power,ground, I2C lines and write protect lines wired. The external address lines (A0,A1,A2) are not connected as those address bits are sent with the I2C data.

The I2C address of the chip is 0xA0 and it has 0x800 memory locations. When addressing a memory location on the chip the top three bits in the address 0x700 are shifted into bits 3,2 & 1 of 0xA0 when the chip is addressed via I2C. The bottom bits of the address 0xFF are sent as the next byte on I2C to set the page address of the memory array. The chip also has a burst mode where you can set up the address for read or write then send or read the next sequential 8 bytes of data in one operation. The chip has a 10 ms write cycle time. That is on a write the chip is busy for 10 ms after the write, note this is for 1 byte or a burst.

The chip supports burst mode where it can read or write up to 16 bytes (in the case of a 24C16) at once with a single address write. The size of the burst is dependent on the start address of the I/O. If we start at say 0 or some multiple of the burst size (16) in this case we can read or write 16. But if the starting address is say 14 the maximum burst would only be 2. The following code calculates the burst size given a starting address and the amount of I/O

```
/*
 * @brief Calculate block size to keep
 *                  on the same page
 *
 * @param bl        - Current block structure
 * @param blocksz   - tentative block size
 * @param len       - Amt left to transfer
 * @param pageaddr  - Start address
 *
 * @return updated block structure to do transfer on same page
 */
static BLOCKS *PageFixer( BLOCKS *bl,uint32 blocksz, uint32 len, uint32 pageaddr )
{
    // (1) Set block size to EEPROM MAX

    bl->blen = EEPROM_BURST_LEN;

    // (2) Is block > amount left ?
    if( bl->blen > len )
        bl->blen = len;

    // (2) Will this value take us off the page ?
    pageaddr &= EEPROM_BURST_PAGE;
    pageaddr += bl->blen;

    // When it goes off the page we need a new block
    if( pageaddr > EEPROM_BURST_LEN )
    {
        bl->blen -= (pageaddr - EEPROM_BURST_LEN );
        bl->blocks++;
    }

    return bl;
}
```

3.1.4.2 Support Code The code to support the chip does two things, first interface with the chip (see below) and second break a read or write into 8 byte or less block so the burst mode can be used. The interface to the chip is via Cypress SCB calls for writes is:

```
// Set up address
status = SCB_I2CMasterSendStart(i2cAddr,SCB_I2C_WRITE_XFER_MODE);
status |= SCB_I2CMasterWriteByte(pageaddr);
// Write data
for(j=0; j<blen;j++)
{
    status |= SCB_I2CMasterWriteByte(buffer[k++]);
}
SCB_I2CMasterSendStop();
CyDelay(WRITE_CYCLE_TIME);  // Wait for write cycle to be done
```

The next two images show a 10 byte write to the chip in two parts an 8 byte burst then a two byte burst. Note the A0 address sent then the starting address (0) then the data.

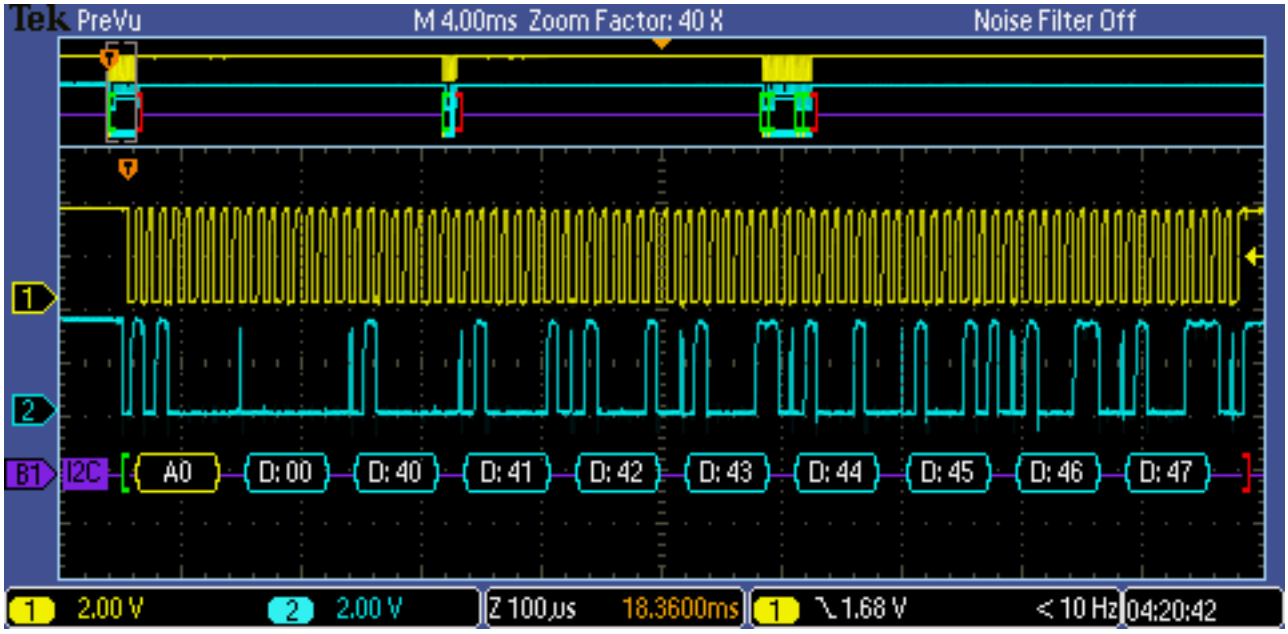

Figure 3: First section of a 10 byte write

This is the second section starting at location at location 8

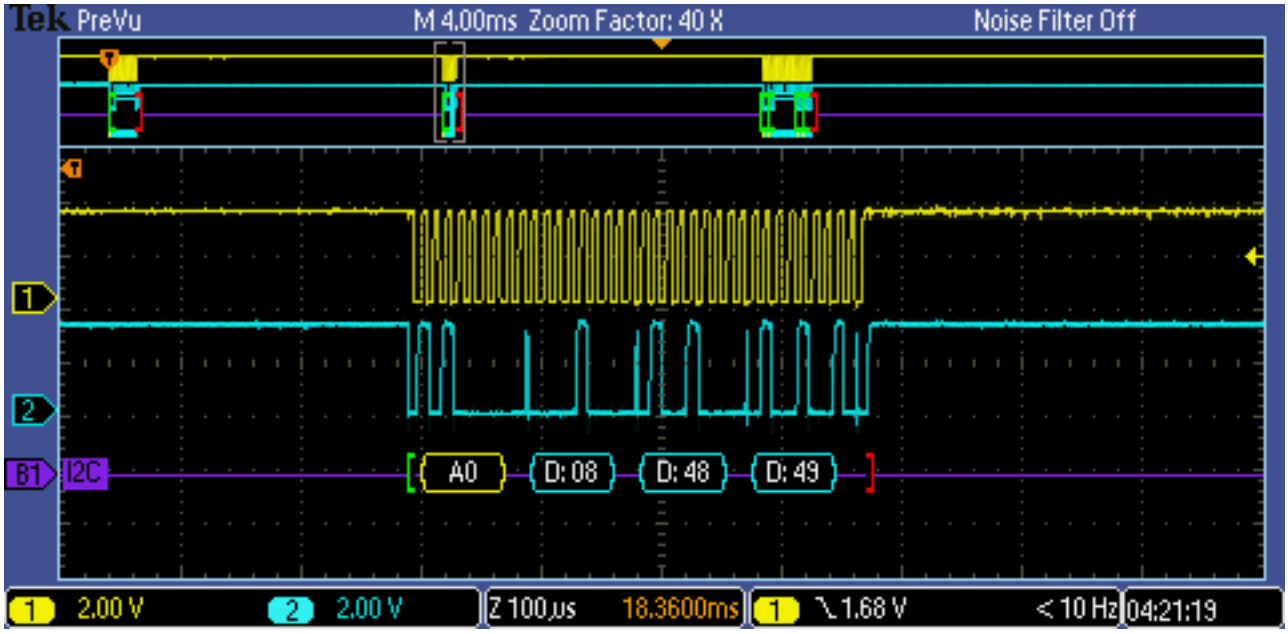

Figure 4: Second section of a 10 byte write

Here shows both transfers (plus the read) note the 10 ms write time separation between the two burst transfers.

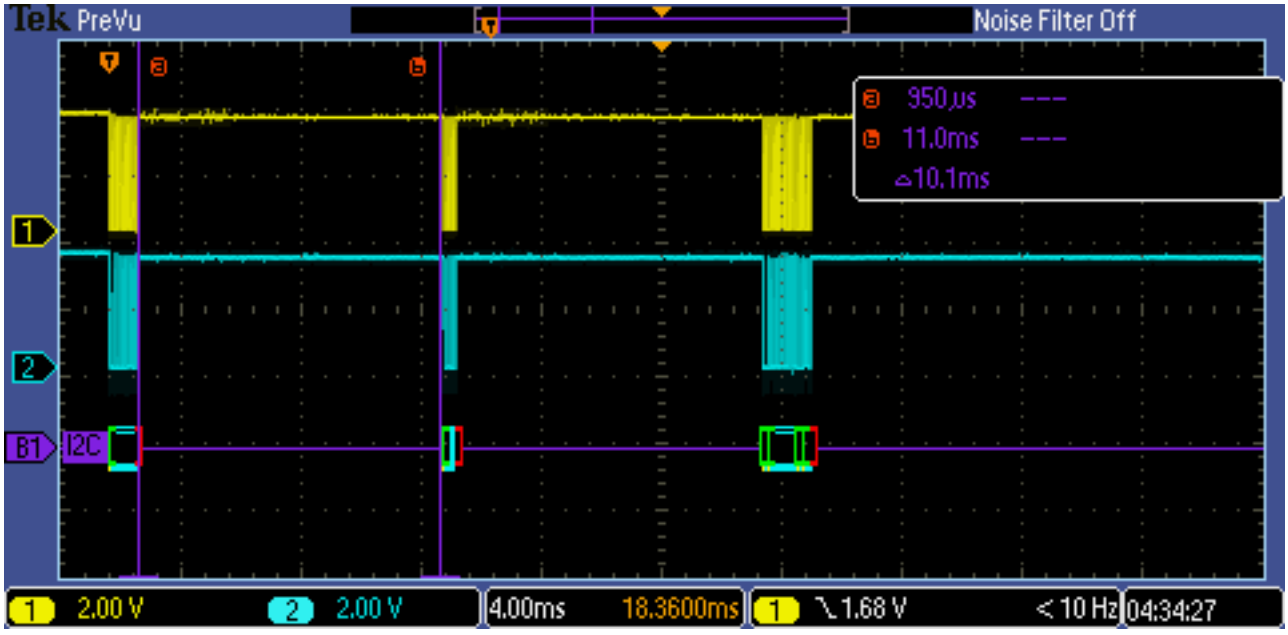

Figure 5: Write time delays

The read code is very similar to the write code but with no delay needed so bursts can be contiguious.

```
// Send Chip/page address & page lcn
    rxbuf[0] = pageaddr;
    status = SCB_I2CMasterWriteBuf(i2cAddr,rxbuf,1,SCB_I2C_MODE_COMPLETE_XFER);

    // Wait till done
    j = 0;
    while( j< 10000 )
    {
        mstatus = SCB_I2CMasterStatus();
        j++;
        if( mstatus & SCB_I2C_MSTAT_WR_CMPLT )
            break;
    }

    // Read the data bytes
    status |= SCB_I2CMasterReadBuf(i2cAddr,ba,bl->blen,SCB_I2C_MODE_REPEAT_START  | SCB_I2C_MODE_COM

    // Wait till done
    j = 0;
    while( j< 10000 )
    {
        mstatus = SCB_I2CMasterStatus();
        j++;
        if( mstatus & SCB_I2C_MSTAT_RD_CMPLT )
            break;
    }
    // Check for OK
    if( status != SCB_I2C_MSTR_NO_ERROR )
        break;
```

The read data is a little more complicated as it does an I2C write with the address then an I2C read to get the data. There is wait code in there so the write is finished before the read starts. And then there is a wait after the read buffer to make sure it is done befoe starting another. The first 8 bytes of the transfer look like this:

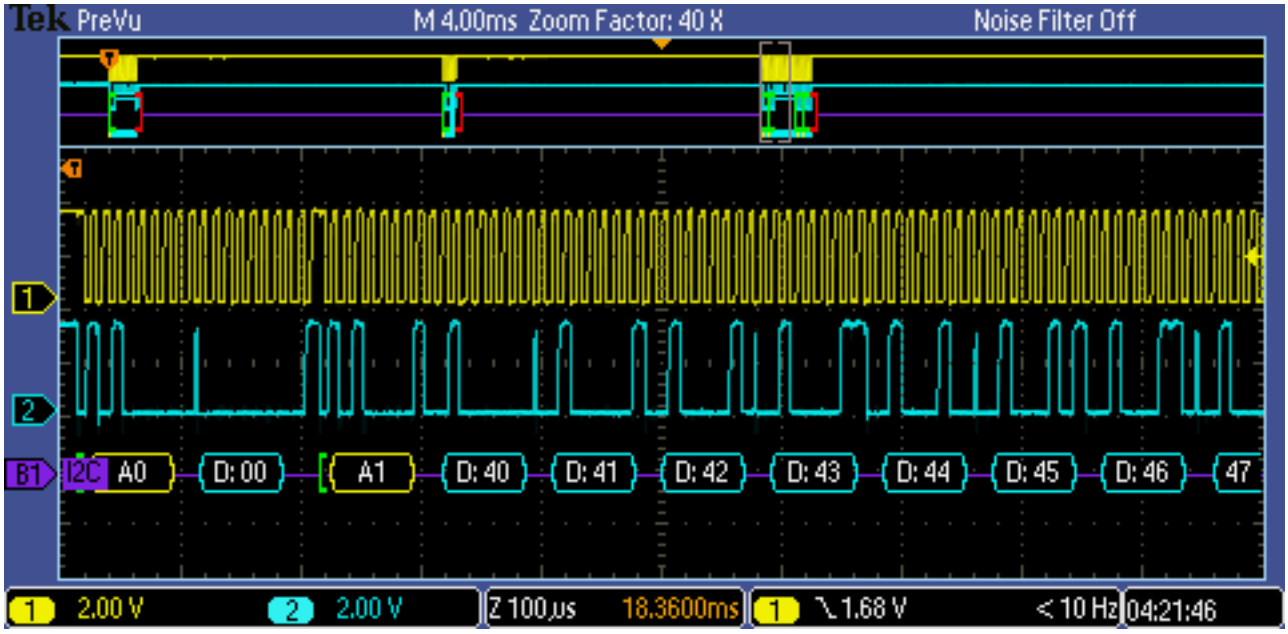

Figure 6: Eight byte burst read

The last two bytes read are done right after this with no delay and starting at addess 8.

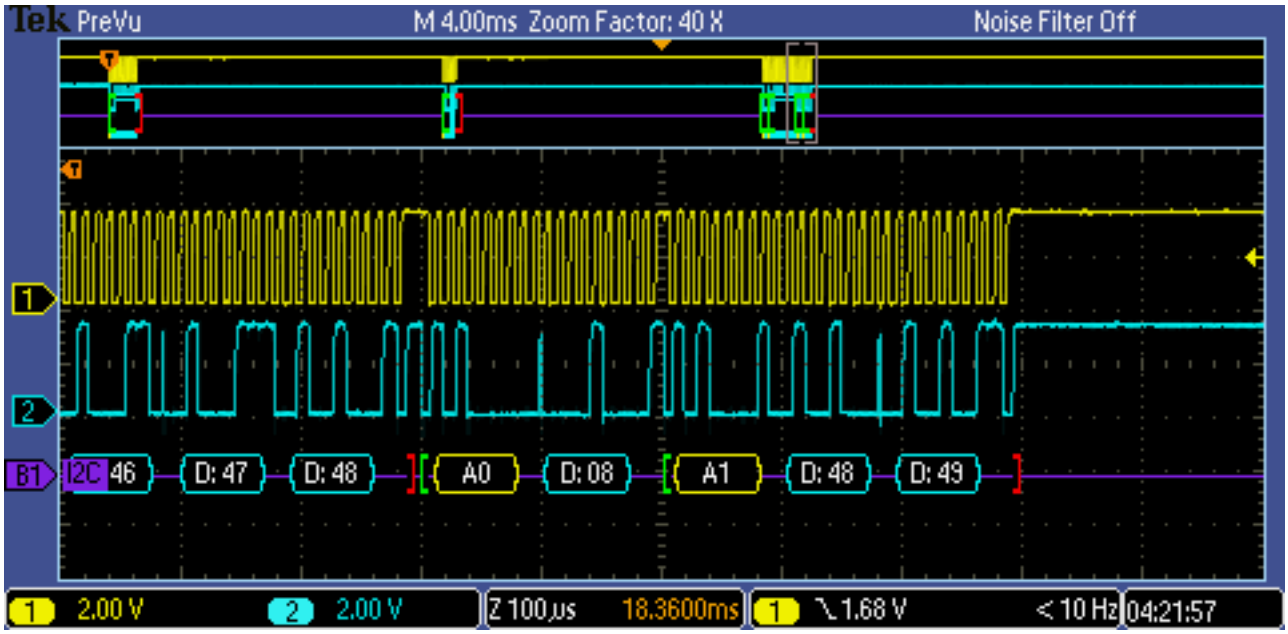

Figure 7: Last 2 bytes of 10 byte read

3.1.5 SysStore EEPROM support

Sitting above the EEPROM is the SysStore module. This code handles the values stored in the EEROM and initializes them on first run, accessed the values and allows them to be changed. It supplies a simple interface for the programmer to set up new EEPROM values or change old ones.

There is a SysStore structure that holds all the data both in EEPROM and a copy in RAM.

```
#define SN_COUNT    10
#define VERSION     2  // Version of program for system
#define VALID_MARK  0x5AA5 + VERSION
// SysStore description
typedef struct  {
    uint16  sysValid;
    uint32  hammerCount;
    uint16  envReadCount;
    uint8   serialNumber[SN_COUNT];
    } SysData;
extern SysData SystemStore;
```

The first uint16 item is a valid mark that incorporates the software version number. So that if the software version changes it will automatically cause the EEPROM to be updated with new values. The rest are a list of the items that are to be preserved. The progammer can add or change these and change the VERSION to update the program.

The program then keeps a copy of this structure in RAM allowing the program quick access to this data. When starting up the main program does this:

```
// Set up EEPROM,
// If not present set default

if( SysStoreValid() == 0 )
{
    val = SysStoreInitialize();
}
val = SysStoreRead();
```

The SysStoreValid checks the valid mark matches the version, if not EEPROM is reset to default. After this the EEPROM values are copied to RAM and used by the program via the extern SystemStore.

When one of these EEPROM values changes the value is then written back to the EEPROM via:

```
SysStoreWriteValue( (uint8 *)&(SystemStore.hammerCount), // Location
                        (uint8 *)&SystemStore.hammerCount,   // Val ptr
                        sizeof(SystemStore.hammerCount));
```

Where the location of the and a pointer to the value are given and that is used to update that value in EEPROM.

4 Transceiver Board

4.1 Hardware

The transceiver board is where all this is brought together. The LPC1114, the EEPROM and the nRF24 board. This will be a simple board that will allow serial control of the nRF24 and with the EEPROM, settings that will survive power cycles. The schematic of the board is:

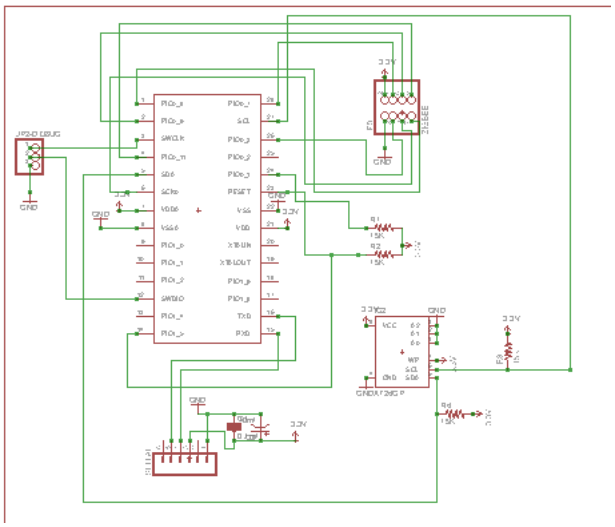

Figure 8: Schematic

As you can see it is powered from the USB<->Serial adapter. The CE and CSN lines are connected to GPIO's on the LPC1114 so they can be programatically controlled. The Eagle board is:

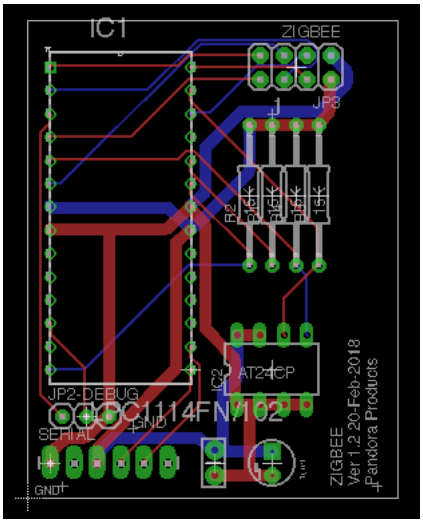

Figure 9: Board

4.2 Software

4.2.1 Introduction

The software is designed to be as flexiable as possible and is largely an interface from the serial port to the SPI connection on the nRF24. As such is is used to send commands to the nRF24 and control the CE line.

4.2.2 Command line

The command line interface is designed to allow full command of all the nRF24 functions and hardware to support this. Input is in two forms, HEX messages and non-HEX messages. Any hex input is converted to a string of unsigned character bytes with any white space removed. Non-HEX input is just input as is captured as is and passed to the handling routines. On HEX input the lenght is > 0 and on non-HEX it is set < 0.

This is the HELP message, output when ? or any unknown command is entered.

```
ZIGBEE Control Program VER:[Feb 23 2018 05:48:51]
Input =><CMD-HEX> <DATA-HEX>
Input => <W/R HEX> <REG> <DATA>
Input -><CMD-nonHEX>
S Task stacks
? Show HELP
P1 Set CE HIGH
P0 Set CE LOW
Q RESET Board
```

The non-HEX message include ? to show help, P0/1 to disable/enable the CE line and S to show stack usage.

```
> p0
CE LOW
> P1
CE HIGH
> s
TASK  STK
--------------------
CMD     00000022
RADIO   0000002B
```

The Q command allows the board to be reset at any time.

The HEX inputs are ment to give commands to the nRF24 over SPI. For example to read a register the read command (0x00) then the register (TX_ADDR) to be read and finally a string of 0 bytes the lenght of the read data is input. The result is the data from the register and the current status byte.

```
> 00 10 0000000000
  ^  ^  ^
  CMD|  |
     REG|
        Output data length 5 bytes
STATUS [0E] DATA E7 E7 E7 E7 E7    Result see [2]command table
> 00 10 0000000000
STATUS [0E] DATA E7 E7 E7 E7 E7 Read XMIT register channels
> 20 10 0102030405
STATUS [0E] DATA 00 00 00 00 00 Set XMIT channels
> 00 10 0000000000
STATUS [0E] DATA 01 02 03 04 05 Read changed values
```

STATUS and CONFIG registers

Two registers inportant in the use of the chip are the CONFIG and STATUS registers. The STATUS register is returned on every I/O operation and the CONFIG register must be written to wake up the chip before other uses.

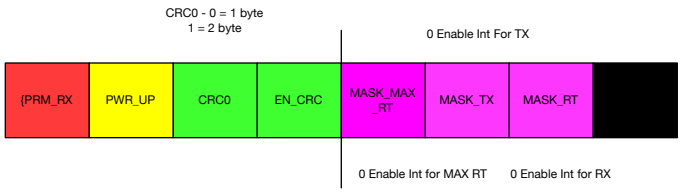

Figure 10: CONFIG register

The status register is:

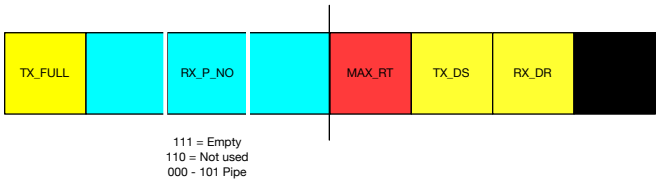

Figure 11: STATUS register

4.2.3 Software Design

The software is designed as two threads running under FreeRTOS. On thread **CONTROL** gets user input and parses the HEX input into an array of unsigned characters. The second thread **RADIO** acts on the input with various functions. A global structure called WORLD is passed to all the threads and contains the data needed by them.

```
#define MAX_INPUT   64
#define MAX_CMD     36
typedef struct
{
    xSemaphoreHandle inputControl;  // Interrupt control
    xSemaphoreHandle radioControl;  // Radio control
    short len;
    uint8 *cmd;
    uint8 *inputBuffer;
    uint8 loop;
}WORLD;
```

The two semaphores are used to synchronize the actions of **RADIO** and **CONTROL** so that after an input **CONTROL** is blocked till the **RADIO** thread handles the input.

The arrays in WORLD are pointers and are linked to static GLOBALs in the main program.

```
static xSemaphoreHandle inputControl; //User input
static xSemaphoreHandle radioControl; // Radio operation
static uint8 cmd[MAX_CMD];
static uint8 inputBuffer[MAX_INPUT];
static WORLD w;.
```

On startup the RADIO thread is unlocked and allowed to run the setup for the nRF24. This does a delay to stabalize the nRF24 then sends a 0x0c to the CONFIG register to allow more commands to be sent.

There is a file RF24.c which contains the SPI routines for sending commands to the nRF24. Any nRF24 additiional code should be added here.

At present the EEPROM is not used but the code supporting is in the project.

References

[1] December 2017.

[2] Nordic Semiconductor. *Single Chip 2.4GHz Transceiver*. Nordic Semiconductor, 7004 Trondheim, Norway, 2008 edition edition, September 2008.