NRF Based Sensor Node - Code Documentation

Sachin Jadhav

August 5, 2018

1 Introduction

This Code Documenatation contains Documentation of three libraries

1. GPIO
2. SPI
3. RF24

2 Code Description

- GPIO.cpp

1. void Set_Pin (unsigned int num, unsigned int dir)
   To set a particular pin as input (IN) or output (OUT) .e.g Set_pin(13,OUT)

```
void Set_pin (unsigned int num, unsigned int dir )
{
        if (dir == OUT )
        {
    if (num >=8 )//check number for port selection
    //set bit corresponding that pin number to make it output
                DDRB |= (1<<(num-8));
                else
                DDRD |= (1<<num);
        }
        else
        {
                if (num >=8 )
        //reset bit corresponding that pin numberto make it input
                DDRB |= (0<<(num-8));
                else
                DDRD |= (0<<num);
        }
}
```

2. void Write_Digital (unsigned char , unsigned char)
   To set a particular output pin as a high or low by giving pin number and value.
   e.g. Write_Digital(13,high);

```
void Write_Digital (unsigned int num, bool value)
{
   //cli();
    if (value  == high )
   {
    if (num >= 8 )
    //set bit in PORTB register to make that pin high
           PORTB |= (1<<(num-8));//OR mask
    else
          PORTD |= (1<<num);
   }
   else
    {
      if (num >=8 )
      //reset bit in PORTB register to make that pin low
           PORTB &= ~(1<<(num-8));// and mask
      else
          PORTD &= ~(1<<num);
    }
   // sei();
}
```

3. void mosfet_on()
   Turn on mosfet which is connected on D8 pin . Which gives supply voltage to NRF and ADC channels

```
void mosfet_on()
{
        DDRB |= 0x01;//mosfet D8 pin
        PORTB |= 0x00;// write 0 to ON P channel MOSFET
}
```

4. void mosfet_off()
   Turn off mosfet which is connected on D8 pin . Which turn off the supply voltage to NRF and ADC channels

```
void mosfet_off()
{

        PORTB &= 0xFE;// write 1to OFF P channel MOSFET
}
```

5. void UART_Init(); //initilzation of uart
   To initialize a UART for serial Communication with default 9600 baud rate

```
void UART_Init()
{
  UCSR0B = 0x00;
  UCSR0A = 0x00;
  UCSR0C|= (1<<UCSZ1) | (1<<UCSZ0)|(1<< USBS0);
  // Asynchronous mode 8-bit data and 1-stop bit
 // UBRR0L = 103;// 16MHZ for 9600 bps
  UBRR0L = 51;
  // 8MHZ Crystal set baud rate 9600(51),4800(103),2400(206)
  for serial transmission with external crystal oscilator at 8.0 MHz
  UBRR0H = 0;
  UCSR0B |= (1<<RXEN) | (1<<TXEN);// Enable Receiption and Transmission
}
```

6. char UART_Read();
   This function checks for any data available on Rx bufer of Atmega328p And if availble Reads 8 bit data in Rxbuffer and returns in char type data.

```
////// to read one byte
char UART_Read()
{
        while (!(UCSR0A & (1<<RXC0))); /* Wait until data exists. */
        //loop_until_bit_is_set(UCSR0A, RXC0);
        return UDR0;
}
```

7. void UART_Transmit(unsigned char data);
   Transmitts single unsigned char data through UART

```
void UART_Transmit(unsigned char data )
{
 UCSR0A = 1<<TXC0;
/* Wait for empty transmit buffer */
while ( !( UCSR0A & (1<<UDRE0)) );
/* Put data into buffer, sends the data */
UDR0 = data;
while (!(UCSR0A & (1<<TXC0)));//wait for tarnsmission to complete
}
```

8. void UART_Printf(const char *text)
   Sends String to serial i.e. to print a string on serial monitor without a newline character

```
void UART_Printf(const char *text)
{
  unsigned char i=0;
  while (!(*(text+i) == '\0'))
  {
    UART_Transmit(*(text+i));
    i++;
  }
}
```

9. void UART_Printfln(const char *text)
   Sends String to serial i.e. to print a string on serial monitor with new line

```
void UART_Printfln(const char *text)
{
  unsigned char i=0;
  while (!(*(text+i) == '\0'))
  {
    UART_Transmit(*(text+i));
    i++;
  }
  UART_Transmit('\n');
}
```

10. void UART_Print_Num(unsigned int n)
    To print unsigned int number on serial monitor

```
/// to print  unsigned int number
void UART_Print_Num(unsigned int n)
{
        unsigned char ch=0;
        ch= n/10000+0x30;
```

```
        if(!(ch==0x30))
                UART_Transmit(ch);
        n=n%10000;
        ch=n/1000+0x30;
        if(!(ch==0x30))
                UART_Transmit(ch);
        n=n%1000;
        ch=n/100+0x30;
        if(!(ch==0x30))
                 UART_Transmit(ch);
        n=n%100;
        ch=n/10+0x30;
        if(!(ch==0x30))
                UART_Transmit(ch);
        ch=(n%10)+0x30;
    UART_Transmit(ch);
    UART_Transmit('\n');
}
```

11. void UART_Print_Numchar(unsigned char n)
    To print unsigned char number on serial monitor

```
/// to print unsigned char number
void UART_Print_Numchar(unsigned char n)
{
        unsigned char ch=0;
        ch=n/100+0x30;
        UART_Transmit(ch);
        n=n%100;
        ch=n/10+0x30;
        UART_Transmit(ch);
        ch=(n%10)+0x30;
        UART_Transmit(ch);
        UART_Transmit('\n');
}
```

12. void timer0_init ();
    TIMER 0 is 8 bit timer
    Timer0 is used in CTC mode for delay function which creates delay of 1 millisecond using TIMERO ISR

```
////timer0 in CTC mode with interrupt
void timer0_init ()
{
        //set timer0 interrupt at 1kHz
        TCCR0A = 0;
        TCCR0B = 0;
        TCNT0  = 0;
        // set compare match register for 1khz increments
        OCR0A = 124;// = (8*10^6) / (1000*64) - 1 (must be <256)
         // turn on CTC mode
         TCCR0A |= (1 << WGM01);
         // Set CS01 and CS00 bits for 64 prescaler
         TCCR0B |= (1 << CS01) | (1 << CS00);
         // enable timer compare interrupt
         TIMSK0 |= (1 << OCIE0A);
}
```

13. ISR(TIMER0_COMPA_vect)

This is Interrupt Sub routine for TIMER 0 interrupt which updates global *cnt* variable for getting current millis value in the running program.

```
// ISR for timer 0
ISR(TIMER0_COMPA_vect)
{
        cnt++;//incremnts milisecond count variable
};
//
```

14. unsigned int millis()
    This function gives current execution time of a program in millisecond
    Returns a Integar variable in between 0 to 65535.

```
/// returns millis count value
unsigned int millis()
{
        //timer0_init();
        return cnt;
}
```

15. void power_down (unsigned int s_time)
    API function for setting u-controller in deep sleep mode for given time in minutes
    This function puts MCU in Power_down mode i.e External Crystal is off , along with for reducing power consumption it Disables Brounout Detection , Disables digital input buffers on ADC pins
    The MCU is configured so as to wake up after predefined timeout using Watchdog timer . Which uses internal 128KHz crystal

```
void power_down (unsigned int s_time)
{
        //for(int i=0;i<20;i++)
        //{
                //Set_pin(i,OUT);
                //
        //}
        //setup watchdog timer for 8s
        // comment below line if delay required is in seconds
         s_time = s_time*8;//as 1 for loop 8 sec sleep factor for 1 minute sleep
        WDTCSR = (24);//change WDCE and WDE also resets
         WDTCSR = (33);//set prescalar for 8 sec timeout which max timeout
    // uncomment below line if delay required in seconds
        // WDTCSR = 0x06; // set prescaler for 1 second timeout
        //WDTCSR |=(1<<6);//enable interrupt mode WDIE set
        //Disable ADC clear ADEN bit
    //dont forget to set this bit while using ADC in main code
    ADCSRA &= ~(1<<7);
         DIDR0=0x3F;//disable digital input buffers
         DIDR1=0x03;//Disabled  AIN Digital Input Disable
     \\It reduces power consumption in digital input buffers
        // select sleep mode using SMCR.SM[2:0]
        SMCR |=(1<<2);//power down mode   010
        SMCR |=1;//enable sleep SMCR.SE set
        for (int i=0;i<s_time;i++)//gives extended timeout for 8*s_time sec
        {
                //BODS Disable
                // disable Brounout detection (BOD)
                MCUCR|=(3<<5);
                MCUCR = (MCUCR &~(1<<5)) | (1<<6);
```

```
                __asm__ __volatile__("sleep");//executes A sleep instruction
        }
}
```

16. ISR(WDT_vect)
    This is Interrupt Subroutine for Watchdog timer which do nothing .Just used for Extending sleep timeout.

```
// ISR For WATCHDOG TIMER
ISR(WDT_vect)
{

};
```

17. unsigned int adcinit(int channel)
    This function do the Initialization of ADC and returns ADC conversion result

```
unsigned int adcinit(int channel)
{
        unsigned int result=0;
        float moisture=0;
        ADMUX = ADMUX & 0x00;// AND masking to selct a particular channel
        ADMUX = _BV(REFS0); // Select ADC refrence voltage as AVCC
    // Selects ADC input channel number on which signal is connected
        ADMUX = ADMUX | channel;
        _delay_ms(2); // Wait for Vref to settle
     // Convert by Enabling & staring the Conversion
        ADCSRA |= _BV(ADSC) + _BV(ADEN);
        while (bit_is_set(ADCSRA,ADSC)); // wait till conversion Complete
        result = ADCL;// Read ADC result lower bytes
        result |= ADCH<<8;// Read ADC result higher bytes
        return result; // return unsigned int 16 bit ADC Conversion value
}
```

18. uint8_t get_batt_voltage()
    This Function reads the ADC channel number 5 .
    On channel number 5 battery voltage is connected . Returns the single byte result of battery voltage.

```
uint8_t get_batt_voltage()
{
        unsigned int result=0;
        float res = 3.3/1024;//resolution
        float voltage;
        // takes multiple values to get correct value
        for(int i=0; i<5;i++)
        {
                result = result + adcinit(5);
        }
        // To get actual battery voltage value  i.e. from 0 to 4.2 max
        voltage =  ((result/5)*res)/0.781;
    //calculation logic of external voltage divider network
        voltage = voltage * 10;// multiply by 10 to get result into integar value
  //4.2 *10 =42 to send single byte
        return (uint8_t)abs(voltage);
}
```

19. uint8_t get_temp(int ch)
    This Function reads the ADC channel number specified on which temperature sensor is

connected.
Returns single byte result temperature value

```
uint8_t get_temp(int ch)
{
        unsigned int result=0;
        float res = 3.3/1024;
        float temp;
    // takes multiple values to get coorect value
        for(int i=0; i<5;i++)
        {
                result = adcinit(ch);
        }

        temp = 186 - 0.3905*result;// temp. sensor equation form datasheet
        //voltage = voltage * 10;
        UART_Print_Num((unsigned int)temp);
        return (uint8_t)abs(temp);
}
```

20. uint8_t get_soil_moisture()
    This Function reads the ADC channel number 0 on which default soil moisture sensor is connected.
    Returns single byte result of moisture content percentage value

```
uint8_t get_soil_moisture()
{
        unsigned int result=0;
        float moisture;
        for(int i=0; i<5;i++)
        {
                result = result + adcinit(0);
        }
        UART_Print_Num(result);
        moisture = - 0.128 * (result/5) + 122; //curvefit
        //Serial.println(moisture); //moisture percentage
        return (uint8_t)abs(moisture);
}
```

21. void getdata()
    Function to collect all sensors data for transmission.

```
void getdata()
{
        data[0] = get_batt_voltage();//battery voltage
        //data[1] = 75;
        data[1] =  get_soil_moisture();
        //data[1] = get_temp();//temp value from temp sensor
        data[2] = 29;//humidity value from DHTsensor
}
```

- SPI library
  This library has two files SPI.h and SPI.cpp
  In SPI.h clas SPIClass is defined with its all methods.
  In SPI.cpp it has API's related initilsation of SPI for connected module.

  1. SPI.h

```
#ifndef _SPI_H_INCLUDED
#define _SPI_H_INCLUDED
// for disbaling name transformation  of function form C file to cpp file
// Include GPIO library to use Set_pin , Write_Digital methods from that library
extern "C"
{
        #include "gpio2.h"
};
// SPI_HAS_TRANSACTION means SPI has beginTransaction(), endTransaction(),
#define SPI_HAS_TRANSACTION 1
class SPIClass
{
public:
  // Initialize the SPI library
  static void begin();
  // Before using SPI.transfer() or asserting chip select pins,
  // this function is used to gain exclusive access to the SPI bus
  // and configure the correct settings.
  inline static void beginTransaction()
   {
    SPCR = 0x50;// fosc/4 (2MHz), mode 0,master mode ,spi enable ,MSB first
    SPSR = 0x00;// no double data rate
  }

  // Write to the SPI bus (MOSI pin) and also receive (MISO pin)
  inline static uint8_t transfer(uint8_t data)
  {
    SPDR = data;
    /*
     * The following NOP introduces a small delay that can prevent the wait
     * loop form iterating when running at the maximum speed. This gives
     * about 10% more speed, even if it seems counter-intuitive. At lower
     * speeds it is unnoticed.
     */
    asm volatile("nop");
    while (!(SPSR & _BV(SPIF))) ; // wait till data is sent successfully
    return SPDR;
  }

  inline static void transfer(void *buf, size_t count)
   {
    if (count == 0) return;
    uint8_t *p = (uint8_t *)buf;
    SPDR = *p;
    while (--count > 0) {
      uint8_t out = *(p + 1);
      while (!(SPSR & _BV(SPIF))) ;
      uint8_t in = SPDR;
      SPDR = out;
      *p++ = in;
    }
    while (!(SPSR & _BV(SPIF))) ;// wait till data is successfully sent
    *p = SPDR;
  }
  // After performing a group of transfers and releasing the chip select
  // signal, this function allows others to access the SPI bus
  inline static void endTransaction(void)
  {
```

```
  }
};
// This class is used in SPI.cpp for writing begin method
extern SPIClass SPI;
#endif
```

2. SPI.cpp

```
\\ Include Necessary header files
\\ Include SPI.h to use its methods to begin SPI Communication
#include "SPI.h"
// to use variables like uint_8, uint_16 etc
#include <stdint-gcc.h>
// Object of SPIclass from SPI.h file
SPIClass SPI;
// Public method defination to intialise the SPI communication
for connected module
void SPIClass::begin()
{
        // set SS pin as output to make microcontroller as master
    Set_pin(SS, OUT);
    // Warning: if the SS pin ever becomes a LOW INPUT then SPI
    // automatically switches to Slave, so the data direction of
    // the SS pin MUST be kept as OUT.

    SPCR |= _BV(MSTR);// selects master mode
    SPCR |= _BV(SPE);//Enables SPI communiation

    // Set direction register for SCK and MOSI pin.
    // MISO pin automatically overrides to INPUT for SPI as master.
    // By doing this AFTER enabling SPI, we avoid accidentally
    // clocking in a single bit since the lines go directly
    // from "input" to SPI control.
    Set_pin(SCK, OUT);// make SCK pin as output pin
    Set_pin(MOSI, OUT);// make MOSI pin as output pn
}
```

3. RF24
   RF24 library contains 4 files
   - RF24config.h
   - nRF24L01.h
   - RF24.h
   - RF24.cpp

   (a) RF24config.h
   This file includes necessary header files for RF24.h and also defines some macros to use in RF24.h file.

```
#ifndef __RF24_CONFIG_H__
#define __RF24_CONFIG_H__
// following used by the method forRF24 object
// to find min and max size of paylaod (data) size
  #define rf24_max(a,b) (a>b?a:b)
  #define rf24_min(a,b) (a<b?a:b)
// This is used for SPI transactions in RF24 library API
  #if defined SPI_HAS_TRANSACTION && !defined SPI_UART && !defined SOFTSPI
    #define RF24_SPI_TRANSACTIONS
```

```
  #endif
  #include "SPI.h"
  #define _SPI SPI // SPI is object in SPI.h and _SPI is object in RF24
  // INcludes necessary header files
  #include <stdint.h>
  #include <stdio.h>
  #include <string.h> // to perform operation on string
  #define _BV(x) (1<<(x))
  //Following macros are used to access program/flash memory
   typedef uint16_t prog_uint16_t;
   #define PSTR(x) (x)
   #define printf_P printf
   #define strlen_P strlen
   // macro for  program memory access
   #define PROGMEM
   #define pgm_read_word(p) (*(p))
   #define PRIPSTR "%s"
#endif // __RF24_CONFIG_H__
```

(b) nRF24L01.h
This header file has declaration of macros for all registers in NRF24L01+ .

```
/* Memory Map */
#define NRF_CONFIG  0x00
#define EN_AA       0x01
#define EN_RXADDR   0x02
#define SETUP_AW    0x03
#define SETUP_RETR  0x04
#define RF_CH       0x05
#define RF_SETUP    0x06
#define NRF_STATUS  0x07
#define OBSERVE_TX  0x08
#define CD          0x09
#define RX_ADDR_P0  0x0A
#define RX_ADDR_P1  0x0B
#define RX_ADDR_P2  0x0C
#define RX_ADDR_P3  0x0D
#define RX_ADDR_P4  0x0E
#define RX_ADDR_P5  0x0F
#define TX_ADDR     0x10
#define RX_PW_P0    0x11
#define RX_PW_P1    0x12
#define RX_PW_P2    0x13
#define RX_PW_P3    0x14
#define RX_PW_P4    0x15
#define RX_PW_P5    0x16
#define FIFO_STATUS 0x17
#define DYNPD       0x1C
#define FEATURE     0x1D

/* Bit Mnemonics */
#define MASK_RX_DR  6
#define MASK_TX_DS  5
#define MASK_MAX_RT 4
#define EN_CRC      3
#define CRCO        2
#define PWR_UP      1
#define PRIM_RX     0
#define ENAA_P5     5
#define ENAA_P4     4
```

```
#define ENAA_P3             3
#define ENAA_P2             2
#define ENAA_P1             1
#define ENAA_P0             0
#define ERX_P5              5
#define ERX_P4              4
#define ERX_P3              3
#define ERX_P2              2
#define ERX_P1              1
#define ERX_P0              0
#define AW                  0
#define ARD                 4
#define ARC                 0
#define PLL_LOCK            4
#define RF_DR               3
#define RF_PWR              6
#define RX_DR               6
#define TX_DS               5
#define MAX_RT              4
#define RX_P_NO             1
#define TX_FULL             0
#define PLOS_CNT            4
#define ARC_CNT             0
#define TX_REUSE            6
#define FIFO_FULL           5
#define TX_EMPTY            4
#define RX_FULL             1
#define RX_EMPTY            0
#define DPL_P5              5
#define DPL_P4              4
#define DPL_P3              3
#define DPL_P2              2
#define DPL_P1              1
#define DPL_P0              0
#define EN_DPL              2
#define EN_ACK_PAY          1
#define EN_DYN_ACK          0

/* Instruction Mnemonics */
#define R_REGISTER          0x00
#define W_REGISTER          0x20
#define REGISTER_MASK       0x1F
#define ACTIVATE            0x50
#define R_RX_PL_WID         0x60
#define R_RX_PAYLOAD        0x61
#define W_TX_PAYLOAD        0xA0
#define W_ACK_PAYLOAD       0xA8
#define FLUSH_TX            0xE1
#define FLUSH_RX            0xE2
#define REUSE_TX_PL         0xE3
#define NOP                 0xFF

/* Non-P omissions */
#define LNA_HCURR   0

/* P model memory Map */
#define RPD                 0x09
#define W_TX_PAYLOAD_NO_ACK  0xB0

/* P model bit Mnemonics */
```

```
#define RF_DR_LOW   5 //set daat rate of transmission
#define RF_DR_HIGH  3
#define RF_PWR_LOW  1 // for setting nrf in power down
#define RF_PWR_HIGH 2 // for powering up the NRF24L01+ module
```

(c) RF24.h

In this library a class RF24 is written with declaration of public ,protected ,private data members and methods.

```
/**
* Includes necessary header files
*/
#include <stdint-gcc.h>
#include "nRF24L01.h"
#include "RF24config.h"
#ifndef __RF24_H__
#define __RF24_H__

/**
 * @file RF24.h
 *
 * Class declaration for RF24 and helper enums
 */
/**
 * Power Amplifier level.
 *
 * For use with setPALevel()
 */
typedef enum { RF24_PA_MIN = 0,RF24_PA_LOW, RF24_PA_HIGH, RF24_PA_MAX, RF24_PA_ERROR } rf

/**
 * Data rate.  How fast data moves through the air.
 *
 * For use with setDataRate()
 */
typedef enum { RF24_1MBPS = 0, RF24_2MBPS, RF24_250KBPS } rf24_datarate_e;

/**
 * CRC Length.  How big (if any) of a CRC is included.
 *
 * For use with setCRCLength()
 */
typedef enum { RF24_CRC_DISABLED = 0, RF24_CRC_8, RF24_CRC_16 } rf24_crclength_e;

class RF24
{
private:
  uint16_t ce_pin; /**< "Chip Enable" pin, activates the RX or TX role */
  uint16_t csn_pin; /**< SPI Chip select */
  uint16_t spi_speed; /**< SPI Bus Speed */
  bool p_variant; /* False for RF24L01 and true for RF24L01P */
  uint8_t payload_size; /**< Fixed size of payloads */
  bool dynamic_payloads_enabled; /**< Whether dynamic payloads are enabled. */
  uint8_t pipe0_reading_address[5]; /**< Last address set on pipe 0 for reading. */
  uint8_t addr_width; /**< The address width to use - 3,4 or 5 bytes. */

// this function are inherited from SPI.h library
protected:
  /**
   * SPI transactions
   *
```

```
   * Common code for SPI transactions including CSN toggle
   *
   */
  // called most oftenly for every data transmission and receiption
  // To get better speed they are made inline
  // begins SPI transaction by making CSN low
  inline void beginTransaction();

  // Ends SPI transaction by making CSN low
  inline void endTransaction();

// ALL public method to configure RF24 object are declared here.
public:

  /**
   * @name Primary public interface
   *
   *  These are the main methods you need to operate the chip
   */
  /**@{*/

  /**
   * ATMEGA328p Constructor
   *
   * Creates a new instance of this driver.  Before using, you create an instance
   * and send in the unique pins that this chip is connected to.
   *
   * @param _cepin The pin attached to Chip Enable on the RF module
   * @param _cspin The pin attached to Chip Select
   */
  RF24(uint16_t _cepin, uint16_t _cspin);

  /**
   * Begin operation of the chip
   *
   * Call this in setup(), before calling any other methods.
   * @code radio.begin() @endcode
   */
  bool begin(void);
  /**
   * Checks if the chip is connected to the SPI bus
   */
  bool isChipConnected();

  /**
   * Start listening on the pipes opened for reading.
   *
   * 1. Be sure to call openReadingPipe() first.
   * 2. Do not call write() while in this mode, without first calling stopListening().
   * 3. Call available() to check for incoming traffic, and read() to get it.
   *
   * @code
   * Open reading pipe 1 using address CCCECCCECC
   *
   * byte address[] = { 0xCC,0xCE,0xCC,0xCE,0xCC };
   * radio.openReadingPipe(1,address);
   * radio.startListening();
   * @endcode
   */
  void startListening(void);
```

```
/**
 * Stop listening for incoming messages, and switch to transmit mode.
 *
 * Do this before calling write().
 * @code
 * radio.stopListening();
 * radio.write(&data,sizeof(data));
 * @endcode
 */
void stopListening(void);

/**
 * Check whether there are bytes available to be read
 * @code
 * if(radio.available()){
 *    radio.read(&data,sizeof(data));
 * }
 * @endcode
 * @return True if there is a payload available, false if none is
 */
bool available(void);

/**
 * Read the available payload
 *
 * The size of data read is the fixed payload size, see getPayloadSize()
 *
 * @note I specifically chose 'void*' as a data type to make it easier
 * for beginners to use.  No casting needed.
 *
 * @note No longer boolean. Use available to determine if packets are
 * available. Interrupt flags are now cleared during reads instead of
 * when calling available().
 *
 * @param buf Pointer to a buffer where the data should be written
 * @param len Maximum number of bytes to read into the buffer
 *
 * @code
 * if(radio.available()){
 *    radio.read(&data,sizeof(data));
 * }
 * @endcode
 * @return No return value. Use available().
 */
void read( void* buf, uint8_t len );

/**
 * Be sure to call openWritingPipe() first to set the destination
 * of where to write to.
 *
 * This blocks until the message is successfully acknowledged by
 * the receiver or the timeout/retransmit maxima are reached.  In
 * the current configuration, the max delay here is 60-70ms.
 *
 * The maximum size of data written is the fixed payload size, see
 * getPayloadSize().  However, you can write less, and the remainder
 * will just be filled with zeroes.
 *
 * TX/RX/RT interrupt flags will be cleared every time write is called
```

```
   *
   * @param buf Pointer to the data to be sent
   * @param len Number of bytes to be sent
   *
   * @code
   * radio.stopListening();
   * radio.write(&data,sizeof(data));
   * @endcode
   * @return True if the payload was delivered successfully false if not
   */
  bool write( const void* buf, uint8_t len );

  /**
   * New: Open a pipe for writing via byte array. Old addressing format retained
   * for compatibility.
   *
   * Only one writing pipe can be open at once, but you can change the address
   * you'll write to. Call stopListening() first.
   *
   * Addresses are assigned via a byte array, default is 5 byte address length
s   *
   * @code
   *    uint8_t addresses[][6] = {"1Node","2Node"};
   *    radio.openWritingPipe(addresses[0]);
   * @endcode
   * @code
   *   uint8_t address[] = { 0xCC,0xCE,0xCC,0xCE,0xCC };
   *   radio.openWritingPipe(address);
   *   address[0] = 0x33;
   *   radio.openReadingPipe(1,address);
   * @endcode
   * @see setAddressWidth
   *
   * @param address The address of the pipe to open. Coordinate these pipe
   * addresses amongst nodes on the network.
   */

  void openWritingPipe(const uint8_t *address);

  /**
   * Open a pipe for reading
   *
   * Up to 6 pipes can be open for reading at once.  Open all the required
   * reading pipes, and then call startListening().
   *
   * @see openWritingPipe
   * @see setAddressWidth
   *
   * @note Pipes 0 and 1 will store a full 5-byte address. Pipes 2-5 will technically
   * only store a single byte, borrowing up to 4 additional bytes from pipe #1 per the
   * assigned address width.
   * @warning Pipes 1-5 should share the same address, except the first byte.
   * Only the first byte in the array should be unique, e.g.
   * @code
   *    uint8_t addresses[][6] = {"1Node","2Node"};
   *    openReadingPipe(1,addresses[0]);
   *    openReadingPipe(2,addresses[1]);
   * @endcode
   *
   * @warning Pipe 0 is also used by the writing pipe.  So if you open
```

```
 * pipe 0 for reading, and then startListening(), it will overwrite the
 * writing pipe.  Ergo, do an openWritingPipe() again before write().
 *
 * @param number Which pipe# to open, 0-5.
 * @param address The 24, 32 or 40 bit address of the pipe to open.
 */

void openReadingPipe(uint8_t number, const uint8_t *address);


/**
 * Test whether there are bytes available to be read in the
 * FIFO buffers.
 *
 * @param[out] pipe_num Which pipe has the payload available
 *
 * @code
 * uint8_t pipeNum;
 * if(radio.available(&pipeNum)){
 *    radio.read(&data,sizeof(data));
 *    Serial.print("Got data on pipe");
 *    Serial.println(pipeNum);
 * }
 * @endcode
 * @return True if there is a payload available, false if none is
 */
bool available(uint8_t* pipe_num);

/**
 * Check if the radio needs to be read. Can be used to prevent data loss
 * @return True if all three 32-byte radio buffers are full
 */
bool rxFifoFull();

/**
 * Enter low-power mode
 *
 * To return to normal power mode, call powerUp().
 *
 * @note After calling startListening(), a basic radio will consume about 13.5mA
 * at max PA level.
 * During active transmission, the radio will consume about 11.5mA, but this will
 * be reduced to 26uA (.026mA) between sending.
 * In full powerDown mode, the radio will consume approximately 900nA (.0009mA)
 *
 * @code
 * radio.powerDown();
 * avr_enter_sleep_mode(); // Custom function to sleep the device
 * radio.powerUp();
 * @endcode
 */
void powerDown(void);

/**
 * Leave low-power mode - required for normal radio operation after calling powerDown(
 *
 * To return to low power mode, call powerDown().
 * @note This will take up to 5ms for maximum compatibility
 */
void powerUp(void) ;
```

```
/**
* Write for single NOACK writes. Optionally disables acknowledgements/autoretries for
*
* @note enableDynamicAck() must be called to enable this feature
*
* Can be used with enableAckPayload() to request a response
* @see enableDynamicAck()
* @see setAutoAck()
* @see write()
*
* @param buf Pointer to the data to be sent
* @param len Number of bytes to be sent
* @param multicast Request ACK (0), NOACK (1)
*/
bool write( const void* buf, uint8_t len, const bool multicast );

/**
 * This will not block until the 3 FIFO buffers are filled with data.
 * Once the FIFOs are full, writeFast will simply wait for success or
 * timeout, and return 1 or 0 respectively. From a user perspective, just
 * keep trying to send the same data. The library will keep auto retrying
 * the current payload using the built in functionality.
 * @warning It is important to never keep the nRF24L01 in TX mode and FIFO full for mo
 * retransmit is enabled, the nRF24L01 is never in TX mode long enough to disobey this
 * to clear by issuing txStandBy() or ensure appropriate time between transmissions.
 *
 * @code
 * Example (Partial blocking):
 *
 *                    radio.writeFast(&buf,32);  // Writes 1 payload to the buffers
 *                    txStandBy();               // Returns 0 if failed. 1 if
 *
 *                    radio.writeFast(&buf,32);  // Writes 1 payload to the buffers
 *                    txStandBy(1000);           // Using extended timeouts, r
 * @endcode
 *
 * @see txStandBy()
 * @see write()
 * @see writeBlocking()
 *
 * @param buf Pointer to the data to be sent
 * @param len Number of bytes to be sent
 * @return True if the payload was delivered successfully false if not
 */
bool writeFast( const void* buf, uint8_t len );

/**
* WriteFast for single NOACK writes. Disables acknowledgements/autoretries for a single
*
* @note enableDynamicAck() must be called to enable this feature
* @see enableDynamicAck()
* @see setAutoAck()
*
* @param buf Pointer to the data to be sent
* @param len Number of bytes to be sent
* @param multicast Request ACK (0) or NOACK (1)
*/
bool writeFast( const void* buf, uint8_t len, const bool multicast );
```

```
  /**
   * This function extends the auto-retry mechanism to any specified duration.
   * It will not block until the 3 FIFO buffers are filled with data.
   * If so the library will auto retry until a new payload is written
   * or the user specified timeout period is reached.
   * @warning It is important to never keep the nRF24L01 in TX mode and FIFO full for mo
   * retransmit is enabled, the nRF24L01 is never in TX mode long enough to disobey this
   * to clear by issuing txStandBy() or ensure appropriate time between transmissions.
   *
   * @code
   * Example (Full blocking):
   *
   *                    radio.writeBlocking(&buf,32,1000); //Wait up to 1 second to writ
   *                    txStandBy(1000);                   //Wait up to 1 second
   *
//Blocks only until user timeout or success. Data flushed on fail.
   * @endcode
   * @note If used from within an interrupt, the interrupt should be disabled until comp
   * @see txStandBy()
   * @see write()
   * @see writeFast()
   *
   * @param buf Pointer to the data to be sent
   * @param len Number of bytes to be sent
   * @param timeout User defined timeout in milliseconds.
   * @return True if the payload was loaded into the buffer successfully false if not
   */
  bool writeBlocking( const void* buf, uint8_t len, uint32_t timeout );

  /**
   * This function should be called as soon as transmission is finished to
   * drop the radio back to STANDBY-I mode. If not issued, the radio will
   * remain in STANDBY-II mode which, per the data sheet, is not a recommended
   * operating mode.
   *
   * @note When transmitting data in rapid succession, it is still recommended by
   * the manufacturer to drop the radio out of TX or STANDBY-II mode if there is
   * time enough between sends for the FIFOs to empty. This is not required if auto-ack
   * is enabled.
   *
   * Relies on built-in auto retry functionality.
   *
   * @code
   * Example (Partial blocking):
   *
   *                    radio.writeFast(&buf,32);
   *                    radio.writeFast(&buf,32);
   *                    radio.writeFast(&buf,32);  //Fills the FIFO buffers up
   *                    bool ok = txStandBy();     //Returns 0 if failed. 1 if success.
   *
//Blocks only until MAX_RT timeout or success. Data flushed on fail.
   * @endcode
   * @see txStandBy(unsigned long timeout)
   * @return True if transmission is successful
   *
   */
   bool txStandBy();

  /**
   * This function allows extended blocking and auto-retries per a user defined timeout
```

```
   * @code
   *    Fully Blocking Example:
   *
   *                    radio.writeFast(&buf,32);
   *                    radio.writeFast(&buf,32);
   *                    radio.writeFast(&buf,32);   //Fills the FIFO buffers up
   *                    bool ok = txStandBy(1000);  //Returns 0 if failed after 1 second
   *
//Blocks only until user defined timeout or success. Data flushed on fail.
   * @endcode
   * @note If used from within an interrupt, the interrupt should be disabled until comp
   * @param timeout Number of milliseconds to retry failed payloads
   * @return True if transmission is successful
   *
   */
   bool txStandBy(uint32_t timeout, bool startTx = 0);

  /**
   * Write an ack payload for the specified pipe
   *
   * The next time a message is received on @p pipe, the data in @p buf will
   * be sent back in the acknowledgement.
   * @see enableAckPayload()
   * @see enableDynamicPayloads()
   * @warning Only three of these can be pending at any time as there are only 3 FIFO bu
   * @note Ack payloads are handled automatically by the radio chip when a payload is re
   * write an ack payload as soon as startListening() is called, so one is available whe
   * @note Ack payloads are dynamic payloads. This only works on pipes 0&1 by default. C
   * enableDynamicPayloads() to enable on all pipes.
   *
   * @param pipe Which pipe# (typically 1-5) will get this response.
   * @param buf Pointer to data that is sent
   * @param len Length of the data to send, up to 32 bytes max.  Not affected
   * by the static payload set by setPayloadSize().
   */
  void writeAckPayload(uint8_t pipe, const void* buf, uint8_t len);

  /**
   * Determine if an ack payload was received in the most recent call to
   * write(). The regular available() can also be used.
   *
   * Call read() to retrieve the ack payload.
   *
   * @return True if an ack payload is available.
   */
  bool isAckPayloadAvailable(void);

  /**
   * Call this when you get an interrupt to find out why
   *
   * Tells you what caused the interrupt, and clears the state of
   * interrupts.
   *
   * @param[out] tx_ok The send was successful (TX_DS)
   * @param[out] tx_fail The send failed, too many retries (MAX_RT)
   * @param[out] rx_ready There is a message waiting to be read (RX_DS)
   */
  void whatHappened(bool& tx_ok,bool& tx_fail,bool& rx_ready);

  /**
```

```
 * Non-blocking write to the open writing pipe used for buffered writes
 *
 * @note Optimization: This function now leaves the CE pin high, so the radio
 * will remain in TX or STANDBY-II Mode until a txStandBy() command is issued. Can be
 * if writing multiple payloads at once.
 * @warning It is important to never keep the nRF24L01 in TX mode with FIFO full for m
 * retransmit/autoAck is enabled, the nRF24L01 is never in TX mode long enough to diso
 * to clear by issuing txStandBy() or ensure appropriate time between transmissions.
 *
 * @see write()
 * @see writeFast()
 * @see startWrite()
 * @see writeBlocking()
 *
 * For single noAck writes see:
 * @see enableDynamicAck()
 * @see setAutoAck()
 *
 * @param buf Pointer to the data to be sent
 * @param len Number of bytes to be sent
 * @param multicast Request ACK (0) or NOACK (1)
 * @return True if the payload was delivered successfully false if not
 */
void startFastWrite( const void* buf, uint8_t len, const bool multicast, bool startTx =

/**
 * Non-blocking write to the open writing pipe
 *
 * Just like write(), but it returns immediately. To find out what happened
 * to the send, catch the IRQ and then call whatHappened().
 *
 * @see write()
 * @see writeFast()
 * @see startFastWrite()
 * @see whatHappened()
 *
 * For single noAck writes see:
 * @see enableDynamicAck()
 * @see setAutoAck()
 *
 * @param buf Pointer to the data to be sent
 * @param len Number of bytes to be sent
 * @param multicast Request ACK (0) or NOACK (1)
 *
 */
void startWrite( const void* buf, uint8_t len, const bool multicast );

/**
 * This function is mainly used internally to take advantage of the auto payload
 * re-use functionality of the chip, but can be beneficial to users as well.
 *
 * The function will instruct the radio to re-use the data in the FIFO buffers,
 * and instructs the radio to re-send once the timeout limit has been reached.
 * Used by writeFast and writeBlocking to initiate retries when a TX failure
 * occurs. Retries are automatically initiated except with the standard write().
 * This way, data is not flushed from the buffer until switching between modes.
 *
 * @note This is to be used AFTER auto-retry fails if wanting to resend
 * using the built-in payload reuse features.
 * After issuing reUseTX(), it will keep reending the same payload forever or until
```

```
 * a payload is written to the FIFO, or a flush_tx command is given.
 */
 void reUseTX();

/**
 * Empty the transmit buffer. This is generally not required in standard operation.
 * May be required in specific cases after stopListening() , if operating at 250KBPS d
 *
 * @return Current value of status register
 */
uint8_t flush_tx(void);

/**
 * Test whether there was a carrier on the line for the
 * previous listening period.
 *
 * Useful to check for interference on the current channel.
 *
 * @return true if was carrier, false if not
 */
bool testCarrier(void);

/**
 * Test whether a signal (carrier or otherwise) greater than
 * or equal to -64dBm is present on the channel. Valid only
 * on nRF24L01P (+) hardware. On nRF24L01, use testCarrier().
 *
 * Useful to check for interference on the current channel and
 * channel hopping strategies.
 *
 * @code
 * bool goodSignal = radio.testRPD();
 * if(radio.available()){
 *    Serial.println(goodSignal ? "Strong signal > 64dBm" : "Weak signal < 64dBm" );
 *    radio.read(0,0);
 * }
 * @endcode
 * @return true if signal => -64dBm, false if not
 */
bool testRPD(void) ;

/**
 * Test whether this is a real radio, or a mock shim for
 * debugging.  Setting either pin to 0xff is the way to
 * indicate that this is not a real radio.
 *
 * @return true if this is a legitimate radio
 */
bool isValid() { return ce_pin != 0xff && csn_pin != 0xff; }

 /**
 * Close a pipe after it has been previously opened.
 * Can be safely called without having previously opened a pipe.
 * @param pipe Which pipe # to close, 0-5.
 */
void closeReadingPipe( uint8_t pipe ) ;

 /**
 * Enable error detection by un-commenting #define FAILURE_HANDLING in RF24_config.h
 * If a failure has been detected, it usually indicates a hardware issue. By default t
```

```
 * will cease operation when a failure is detected.
 * This should allow advanced users to detect and resolve intermittent hardware issues
 *
 * In most cases, the radio must be re-enabled via radio.begin(); and the appropriate
 * applied after a failure occurs, if wanting to re-enable the device immediately.
 *
 * Usage: (Failure handling must be enabled per above)
 *  @code
 *  if(radio.failureDetected){
 *    radio.begin();                          // Attempt to re-configure the radio with d
 *    radio.failureDetected = 0;              // Reset the detection value
 *    radio.openWritingPipe(addresses[1]); // Re-configure pipe addresses
 *    radio.openReadingPipe(1,addresses[0]);
 *    report_failure();                       // Blink leds, send a message, etc. to indi
 *  }
 * @endcode
*/
//#if defined (FAILURE_HANDLING)
  bool failureDetected;
//#endif

/**@}*/

/**@}*/
/**
 * @name Optional Configurators
 *
 *  Methods you can use to get or set the configuration of the chip.
 *  None are required.  Calling begin() sets up a reasonable set of
 *  defaults.
 */
/**@{*/

/**
* Set the address width from 3 to 5 bytes (24, 32 or 40 bit)
*
* @param a_width The address width to use: 3,4 or 5
*/

void setAddressWidth(uint8_t a_width);

/**
 * Set the number and delay of retries upon failed submit
 *
 * @param delay How long to wait between each retry, in multiples of 250us,
 * max is 15.  0 means 250us, 15 means 4000us.
 * @param count How many retries before giving up, max 15
 */
void setRetries(uint8_t delay, uint8_t count);

/**
 * Set RF communication channel
 *
 * @param channel Which RF channel to communicate on, 0-125
 */
void setChannel(uint8_t channel);

  /**
 * Get RF communication channel
 *
```

```
 * @return The currently configured RF Channel
 */
uint8_t getChannel(void);

/**
 * Set Static Payload Size
 *
 * This implementation uses a pre-stablished fixed payload size for all
 * transmissions.  If this method is never called, the driver will always
 * transmit the maximum payload size (32 bytes), no matter how much
 * was sent to write().
 *
 * @todo Implement variable-sized payloads feature
 *
 * @param size The number of bytes in the payload
 */
void setPayloadSize(uint8_t size);

/**
 * Get Static Payload Size
 *
 * @see setPayloadSize()
 *
 * @return The number of bytes in the payload
 */
uint8_t getPayloadSize(void);

/**
 * Get Dynamic Payload Size
 *
 * For dynamic payloads, this pulls the size of the payload off
 * the chip
 *
 * @note Corrupt packets are now detected and flushed per the
 * manufacturer.
 * @code
 * if(radio.available()){
 *   if(radio.getDynamicPayloadSize() < 1){
 *     // Corrupt payload has been flushed
 *     return;
 *   }
 *   radio.read(&data,sizeof(data));
 * }
 * @endcode
 *
 * @return Payload length of last-received dynamic payload
 */
uint8_t getDynamicPayloadSize(void);

/**
 * Enable custom payloads on the acknowledge packets
 *
 * Ack payloads are a handy way to return data back to senders without
 * manually changing the radio modes on both units.
 *
 * @note Ack payloads are dynamic payloads. This only works on pipes 0&1 by default. C
 * enableDynamicPayloads() to enable on all pipes.
 */
void enableAckPayload(void);
```

```
/**
 * Enable dynamically-sized payloads
 *
 * This way you don't always have to send large packets just to send them
 * once in a while.  This enables dynamic payloads on ALL pipes.
 *
 */
void enableDynamicPayloads(void);

/**
 * Disable dynamically-sized payloads
 *
 * This disables dynamic payloads on ALL pipes. Since Ack Payloads
 * requires Dynamic Payloads, Ack Payloads are also disabled.
 * If dynamic payloads are later re-enabled and ack payloads are desired
 * then enableAckPayload() must be called again as well.
 *
 */
void disableDynamicPayloads(void);

/**
 * Enable dynamic ACKs (single write multicast or unicast) for chosen messages
 *
 * @note To enable full multicast or per-pipe multicast, use setAutoAck()
 *
 * @warning This MUST be called prior to attempting single write NOACK calls
 * @code
 * radio.enableDynamicAck();
 * radio.write(&data,32,1);  // Sends a payload with no acknowledgement requested
 * radio.write(&data,32,0);  // Sends a payload using auto-retry/autoACK
 * @endcode
 */
void enableDynamicAck();
/**
 * Determine whether the hardware is an nRF24L01+ or not.
 *
 * @return true if the hardware is nRF24L01+ (or compatible) and false
 * if its not.
 */
bool isPVariant(void) ;

/**
 * Enable or disable auto-acknowlede packets
 *
 * This is enabled by default, so it's only needed if you want to turn
 * it off for some reason.
 *
 * @param enable Whether to enable (true) or disable (false) auto-acks
 */
void setAutoAck(bool enable);

/**
 * Enable or disable auto-acknowlede packets on a per pipeline basis.
 *
 * AA is enabled by default, so it's only needed if you want to turn
 * it off/on for some reason on a per pipeline basis.
 *
 * @param pipe Which pipeline to modify
 * @param enable Whether to enable (true) or disable (false) auto-acks
 */
```

```
void setAutoAck( uint8_t pipe, bool enable ) ;

/**
 * Set Power Amplifier (PA) level to one of four levels:
 * RF24_PA_MIN, RF24_PA_LOW, RF24_PA_HIGH and RF24_PA_MAX
 *
 * The power levels correspond to the following OUT levels respectively:
 * NRF24L01: -18dBm, -12dBm,-6dBM, and 0dBm
 *
 * SI24R1: -6dBm, 0dBm, 3dBM, and 7dBm.
 *
 * @param level Desired PA level.
 */
void setPALevel ( uint8_t level );

/**
 * Fetches the current PA level.
 *
 * NRF24L01: -18dBm, -12dBm, -6dBm and 0dBm
 * SI24R1:   -6dBm, 0dBm, 3dBm, 7dBm
 *
 * @return Returns values 0 to 3 representing the PA Level.
 */
 uint8_t getPALevel( void );

/**
 * Set the transmission data rate
 *
 * @warning setting RF24_250KBPS will fail for non-plus units
 *
 * @param speed RF24_250KBPS for 250kbs, RF24_1MBPS for 1Mbps, or RF24_2MBPS for 2Mbps
 * @return true if the change was successful
 */
bool setDataRate(rf24_datarate_e speed);

/**
 * Fetches the transmission data rate
 *
 * @return Returns the hardware's currently configured datarate. The value
 * is one of 250kbs, RF24_1MBPS for 1Mbps, or RF24_2MBPS, as defined in the
 * rf24_datarate_e enum.
 */
rf24_datarate_e getDataRate( void ) ;

/**
 * Set the CRC length
 * <br>CRC checking cannot be disabled if auto-ack is enabled
 * @param length RF24_CRC_8 for 8-bit or RF24_CRC_16 for 16-bit
 */
void setCRCLength(rf24_crclength_e length);

/**
 * Get the CRC length
 * <br>CRC checking cannot be disabled if auto-ack is enabled
 * @return RF24_CRC_DISABLED if disabled or RF24_CRC_8 for 8-bit or RF24_CRC_16 for 16-
 */
rf24_crclength_e getCRCLength(void);

/**
 * Disable CRC validation
```

```
 *
 * @warning CRC cannot be disabled if auto-ack/ESB is enabled.
 */
void disableCRC( void ) ;

/**
* The radio will generate interrupt signals when a transmission is complete,
* a transmission fails, or a payload is received. This allows users to mask
* those interrupts to prevent them from generating a signal on the interrupt
* pin. Interrupts are enabled on the radio chip by default.
*
* @code
*     Mask all interrupts except the receive interrupt:
*
*             radio.maskIRQ(1,1,0);
* @endcode
*
* @param tx_ok  Mask transmission complete interrupts
* @param tx_fail  Mask transmit failure interrupts
* @param rx_ready Mask payload received interrupts
*/
void maskIRQ(bool tx_ok,bool tx_fail,bool rx_ready);

/**
*
* The driver will delay for this duration when stopListening() is called
*
* When responding to payloads, faster devices like ARM(RPi) are much faster than Arduit
* 1. Arduino sends data to RPi, switches to RX mode
* 2. The RPi receives the data, switches to TX mode and sends before the Arduino radio
* 3. If AutoACK is disabled, this can be set as low as 0. If AA/ESB enabled, set to 10
*
* @warning If set to 0, ensure 130uS delayf after stopListening() and before any sends
*/

uint32_t txDelay;

/**
*
* On all devices but Linux and ATTiny, a small delay is added to the CSN toggling func
*
* This is intended to minimise the speed of SPI polling due to radio commands
*
* If using interrupts or timed requests, this can be set to 0 Default:5
*/

uint32_t csDelay;

/**@}*/
/**
 * @name Deprecated
 *
 *  Methods provided for backwards compabibility.
 */
/**@{*/


/**
 * Open a pipe for reading
 * @note For compatibility with old code only, see new function
```

```
  *
  * @warning Pipes 1-5 should share the first 32 bits.
  * Only the least significant byte should be unique, e.g.
  * @code
  *   openReadingPipe(1,0xF0F0F0F0AA);
  *   openReadingPipe(2,0xF0F0F0F066);
  * @endcode
  *
  * @warning Pipe 0 is also used by the writing pipe.  So if you open
  * pipe 0 for reading, and then startListening(), it will overwrite the
  * writing pipe.  Ergo, do an openWritingPipe() again before write().
  *
  * @param number Which pipe# to open, 0-5.
  * @param address The 40-bit address of the pipe to open.
  */
 void openReadingPipe(uint8_t number, uint64_t address);

 /**
  * Open a pipe for writing
  * @note For compatibility with old code only, see new function
  *
  * Addresses are 40-bit hex values, e.g.:
  *
  * @code
  *   openWritingPipe(0xF0F0F0F0F0);
  * @endcode
  *
  * @param address The 40-bit address of the pipe to open.
  */
 void openWritingPipe(uint64_t address);

 /**
  * Empty the receive buffer
  *
  * @return Current value of status register
  */
 uint8_t flush_rx(void);
 /**
  * Read a chunk of data in from a register
  *
  * @param reg Which register. Use constants from nRF24L01.h
  * @param buf Where to put the data
  * @param len How many bytes of data to transfer
  * @return Current value of status register
  */
 uint8_t read_register(uint8_t reg, uint8_t* buf, uint8_t len);

 /**
  * Read single byte from a register
  *
  * @param reg Which register. Use constants from nRF24L01.h
  * @return Current value of register @p reg
  */
 uint8_t read_register(uint8_t reg);



private:

 /**
```

```
  * @name Low-level internal interface.
  *
  *  Protected methods that address the chip directly.  Regular users cannot
  *  ever call these.  They are documented for completeness and for developers who
  *  may want to extend this class.
  */
 /**@{*/

 /**
  * Set chip select pin
  *
  * Running SPI bus at PI_CLOCK_DIV2 so we don't waste time transferring data
  * and best of all, we make use of the radio's FIFO buffers. A lower speed
  * means we're less likely to effectively leverage our FIFOs and pay a higher
  * AVR runtime cost as toll.
  *
  * @param mode HIGH to take this unit off the SPI bus, LOW to put it on
  */
 void csn(bool mode);

 /**
  * Set chip enable
  *
  * @param level HIGH to actively begin transmission or LOW to put in standby.
Please see data sheet
  * for a much more detailed description of this pin.
  */
 void ce(bool level);
  /**
  * Write a chunk of data to a register
  *
  * @param reg Which register. Use constants from nRF24L01.h
  * @param buf Where to get the data
  * @param len How many bytes of data to transfer
  * @return Current value of status register
  */
 uint8_t write_register(uint8_t reg, const uint8_t* buf, uint8_t len);

 /**
  * Write a single byte to a register
  *
  * @param reg Which register. Use constants from nRF24L01.h
  * @param value The new value to write
  * @return Current value of status register
  */
 uint8_t write_register(uint8_t reg, uint8_t value);

 /**
  * Write the transmit payload
  *
  * The size of data written is the fixed payload size, see getPayloadSize()
  *
  * @param buf Where to get the data
  * @param len Number of bytes to be sent
  * @return Current value of status register
  */
 uint8_t write_payload(const void* buf, uint8_t len, const uint8_t writeType);

 /**
  * Read the receive payload
```

```
     *
     * The size of data read is the fixed payload size, see getPayloadSize()
     *
     * @param buf Where to put the data
     * @param len Maximum number of bytes to read
     * @return Current value of status register
     */
    uint8_t read_payload(void* buf, uint8_t len);

    /**
     * Retrieve the current status of the chip
     *
     * @return Current value of status register
     */
    uint8_t get_status(void);
    /**
     * Turn on or off the special features of the chip
     *
     * The chip has certain 'features' which are only available when the 'features'
     * are enabled.  See the datasheet for details.
     */
    void toggle_features(void);

    /**
     * Built in spi transfer function to simplify repeating code repeating code
     */

    uint8_t spiTrans(uint8_t cmd);

    #if defined (FAILURE_HANDLING) || defined (RF24_LINUX)
          void errNotify(void);
    #endif
  };// end of RF24 Class

  #endif // __RF24_H__
```

(d) RF24.cpp
In this file defination of all methods of RF24 object are written in C++.