```
/*
 Copyright (C) 2011 J. Coliz <maniacbug@ymail.com>

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 version 2 as published by the Free Software Foundation.
 */

/*
       Comment from MECH307 group: This file was modified to run on the Arduino Due. Since the "_BV" macro is not available for the
       DUE, "<<" was used instead. Additionally, several functions were commented out since they were not going to be used and we
       didn't want to spend time making them functional on the DUE.
*/

#include "nRF24L01.h"
#include "RF24_config.h"
#include "RF24.h"

/****************************************************************************/

void RF24::csn(int mode)
{
  // Minimum ideal SPI bus speed is 2x data rate
  // If we assume 2Mbs data rate and 16Mhz clock, a
  // divider of 4 is the minimum we want.
  // CLK:BUS 8Mhz:2Mhz, 16Mhz:4Mhz, or 20Mhz:5Mhz
#ifdef ARDUINO
  SPI.setBitOrder(52, MSBFIRST);
  SPI.setDataMode(52, SPI_MODE0);
  SPI.setClockDivider(52, 21);               //was div by 4 to get 4MHZ, for Due, div by 21 to get 4MHZ  //MRH
#endif
  digitalWrite(csn_pin,mode);
}

/****************************************************************************/

void RF24::ce(int level)
{
  digitalWrite(ce_pin,level);
}

/****************************************************************************/

uint8_t RF24::read_register(uint8_t reg, uint8_t* buf, uint8_t len)
{
  uint8_t status;

  csn(LOW);
  status = SPI.transfer(52, R_REGISTER | ( REGISTER_MASK & reg ) );
  while ( len-- )
    *buf++ = SPI.transfer(52, 0xff);
```

```
  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::read_register(uint8_t reg)
{
  csn(LOW);
  SPI.transfer(52, R_REGISTER | ( REGISTER_MASK & reg ) );
  uint8_t result = SPI.transfer(52, 0xff);

  csn(HIGH);
  return result;
}

/****************************************************************************/

uint8_t RF24::write_register(uint8_t reg, const uint8_t* buf, uint8_t len)
{
  uint8_t status;

  csn(LOW);
  status = SPI.transfer(52, W_REGISTER | ( REGISTER_MASK & reg ) );
  while ( len-- )
    SPI.transfer(52, *buf++);

  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::write_register(uint8_t reg, uint8_t value)
{
  uint8_t status;

  IF_SERIAL_DEBUG(printf_P(PSTR("write_register(%02x,%02x)\r\n"),reg,value));

  csn(LOW);
  status = SPI.transfer(52, W_REGISTER | ( REGISTER_MASK & reg ) );
  SPI.transfer(52, value);
  csn(HIGH);

  return status;
}

/****************************************************************************/
```

```
uint8_t RF24::write_payload(const void* buf, uint8_t len)
{
  uint8_t status;

  const uint8_t* current = reinterpret_cast<const uint8_t*>(buf);

  uint8_t data_len = min(len,payload_size);
  uint8_t blank_len = dynamic_payloads_enabled ? 0 : payload_size - data_len;

  //printf("[Writing %u bytes %u blanks]",data_len,blank_len);

  csn(LOW);
  status = SPI.transfer(52, W_TX_PAYLOAD );
  while ( data_len-- )
    SPI.transfer(52, *current++);
  while ( blank_len-- )
    SPI.transfer(52, 0);
  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::read_payload(void* buf, uint8_t len)
{
  uint8_t status;
//  uint32_t* current = reinterpret_cast<uint32_t*>(buf);
  uint8_t* current = reinterpret_cast<uint8_t*>(buf);

  uint8_t data_len = min(len,payload_size);
  uint8_t blank_len = dynamic_payloads_enabled ? 0 : payload_size - data_len;

  //printf("[Reading %u bytes %u blanks]",data_len,blank_len);

  csn(LOW);
  status = SPI.transfer(52, R_RX_PAYLOAD );
  while ( data_len-- )
    *current++ = SPI.transfer(52, 0xff);
  while ( blank_len-- )
    SPI.transfer(52, 0xff);
  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::flush_rx(void)
{
```

```
  uint8_t status;

  csn(LOW);
  status = SPI.transfer(52, FLUSH_RX );
  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::flush_tx(void)
{
  uint8_t status;

  csn(LOW);
  status = SPI.transfer(52, FLUSH_TX );
  csn(HIGH);

  return status;
}

/****************************************************************************/

uint8_t RF24::get_status(void)
{
  uint8_t status;

  csn(LOW);
  status = SPI.transfer(52, NOP );
  csn(HIGH);

  return status;
}

/****************************************************************************/

/*
void RF24::print_status(uint8_t status)
{
  printf_P(PSTR("STATUS\t\t = 0x%02x RX_DR=%x TX_DS=%x MAX_RT=%x RX_P_NO=%x TX_FULL=%x\r\n"),
           status,
           (status & 1<<(RX_DR))?1:0,
           (status & 1<<(TX_DS))?1:0,
           (status & 1<<(MAX_RT))?1:0,
           ((status >> RX_P_NO) & B111),
           (status & 1<<(TX_FULL))?1:0
          );
}
*/             //Block comment since this code shouldn't be needed
```

```
/****************************************************************************/

/*
void RF24::print_observe_tx(uint8_t value)
{
  printf_P(PSTR("OBSERVE_TX=%02x: POLS_CNT=%x ARC_CNT=%x\r\n"),
           value,
           (value >> PLOS_CNT) & B1111,
           (value >> ARC_CNT) & B1111
          );
}
*/      //MRH - not needed

/****************************************************************************/

/*
void RF24::print_byte_register(const char* name, uint8_t reg, uint8_t qty)
{
  char extra_tab = strlen_P(name) < 8 ? '\t' : 0;
  printf_P(PSTR(PRIPSTR"\t%c ="),name,extra_tab);
  while (qty--)
    printf_P(PSTR(" 0x%02x"),read_register(reg++));
  printf_P(PSTR("\r\n"));
}
*/      //MRH - not needed

/****************************************************************************/

/*
void RF24::print_address_register(const char* name, uint8_t reg, uint8_t qty)
{
  char extra_tab = strlen_P(name) < 8 ? '\t' : 0;
  printf_P(PSTR(PRIPSTR"\t%c ="),name,extra_tab);

  while (qty--)
  {
    uint8_t buffer[5];
    read_register(reg++,buffer,sizeof buffer);

    printf_P(PSTR(" 0x"));
    uint8_t* bufptr = buffer + sizeof buffer;
    while( --bufptr >= buffer )
      printf_P(PSTR("%02x"),*bufptr);
  }

  printf_P(PSTR("\r\n"));
}
*/      //MRH - not needed

/****************************************************************************/
```

```
RF24::RF24(uint8_t _cepin, uint8_t _cspin):
  ce_pin(_cepin), csn_pin(_cspin), wide_band(true), p_variant(false),
  payload_size(32), ack_payload_available(false), dynamic_payloads_enabled(false),
  pipe0_reading_address(0)
{
}

/****************************************************************************/

void RF24::setChannel(uint8_t channel)
{
  // TODO: This method could take advantage of the 'wide_band' calculation
  // done in setChannel() to require certain channel spacing.

  const uint8_t max_channel = 127;
  write_register(RF_CH,min(channel,max_channel));
}

/****************************************************************************/

void RF24::setPayloadSize(uint8_t size)
{
  const uint8_t max_payload_size = 32;
  payload_size = min(size,max_payload_size);
}

/****************************************************************************/

uint8_t RF24::getPayloadSize(void)
{
  return payload_size;
}

/****************************************************************************/

static const char rf24_datarate_e_str_0[] PROGMEM = "1MBPS";
static const char rf24_datarate_e_str_1[] PROGMEM = "2MBPS";
static const char rf24_datarate_e_str_2[] PROGMEM = "250KBPS";
static const char * const rf24_datarate_e_str_P[] PROGMEM = {
  rf24_datarate_e_str_0,
  rf24_datarate_e_str_1,
  rf24_datarate_e_str_2,
};
static const char rf24_model_e_str_0[] PROGMEM = "nRF24L01";
static const char rf24_model_e_str_1[] PROGMEM = "nRF24L01+";
static const char * const rf24_model_e_str_P[] PROGMEM = {
  rf24_model_e_str_0,
  rf24_model_e_str_1,
};
static const char rf24_crclength_e_str_0[] PROGMEM = "Disabled";
static const char rf24_crclength_e_str_1[] PROGMEM = "8 bits";
```

```
static const char rf24_crclength_e_str_2[] PROGMEM = "16 bits" ;
static const char * const rf24_crclength_e_str_P[] PROGMEM = {
  rf24_crclength_e_str_0,
  rf24_crclength_e_str_1,
  rf24_crclength_e_str_2,
};
static const char rf24_pa_dbm_e_str_0[] PROGMEM = "PA_MIN";
static const char rf24_pa_dbm_e_str_1[] PROGMEM = "PA_LOW";
static const char rf24_pa_dbm_e_str_2[] PROGMEM = "LA_MED";
static const char rf24_pa_dbm_e_str_3[] PROGMEM = "PA_HIGH";
static const char * const rf24_pa_dbm_e_str_P[] PROGMEM = {
  rf24_pa_dbm_e_str_0,
  rf24_pa_dbm_e_str_1,
  rf24_pa_dbm_e_str_2,
  rf24_pa_dbm_e_str_3,
};

/*
void RF24::printDetails(void)
{
  print_status(get_status());

  print_address_register(PSTR("RX_ADDR_P0-1"),RX_ADDR_P0,2);
  print_byte_register(PSTR("RX_ADDR_P2-5"),RX_ADDR_P2,4);
  print_address_register(PSTR("TX_ADDR"),TX_ADDR);

  print_byte_register(PSTR("RX_PW_P0-6"),RX_PW_P0,6);
  print_byte_register(PSTR("EN_AA"),EN_AA);
  print_byte_register(PSTR("EN_RXADDR"),EN_RXADDR);
  print_byte_register(PSTR("RF_CH"),RF_CH);
  print_byte_register(PSTR("RF_SETUP"),RF_SETUP);
  print_byte_register(PSTR("CONFIG"),CONFIG);
  print_byte_register(PSTR("DYNPD/FEATURE"),DYNPD,2);

  printf_P(PSTR("Data Rate\t = %S\r\n"),pgm_read_word(&rf24_datarate_e_str_P[getDataRate()]));
  printf_P(PSTR("Model\t\t = %S\r\n"),pgm_read_word(&rf24_model_e_str_P[isPVariant()]));
  printf_P(PSTR("CRC Length\t = %S\r\n"),pgm_read_word(&rf24_crclength_e_str_P[getCRCLength()]));
  printf_P(PSTR("PA Power\t = %S\r\n"),pgm_read_word(&rf24_pa_dbm_e_str_P[getPALevel()]));
}
*/      //MRH - not needed
/****************************************************************************/

void RF24::begin(void)
{
  // Initialize pins
  pinMode(ce_pin,OUTPUT);
  pinMode(csn_pin,OUTPUT);

  // Initialize SPI bus
  SPI.begin(52);
```

```
  ce(LOW);
  csn(HIGH);

  // Must allow the radio time to settle else configuration bits will not necessarily stick.
  // This is actually only required following power up but some settling time also appears to
  // be required after resets too. For full coverage, we'll always assume the worst.
  // Enabling 16b CRC is by far the most obvious case if the wrong timing is used - or skipped.
  // Technically we require 4.5ms + 14us as a worst case. We'll just call it 5ms for good measure.
  // WARNING: Delay is based on P-variant whereby non-P *may* require different timing.
  delay( 5 ) ;

  // Set 1500uS (minimum for 32B payload in ESB@250KBPS) timeouts, to make testing a little easier
  // WARNING: If this is ever lowered, either 250KBS mode with AA is broken or maximum packet
  // sizes must never be used. See documentation for a more complete explanation.
  write_register(SETUP_RETR,(B0100 << ARD) | (B1111 << ARC));

  // Restore our default PA level
  setPALevel( RF24_PA_MAX ) ;

  // Determine if this is a p or non-p RF24 module and then
  // reset our data rate back to default value. This works
  // because a non-P variant won't allow the data rate to
  // be set to 250Kbps.
  if( setDataRate( RF24_250KBPS ) )
  {
    p_variant = true ;
  }

  // Then set the data rate to the slowest (and most reliable) speed supported by all
  // hardware.
  setDataRate( RF24_1MBPS ) ;

  // Initialize CRC and request 2-byte (16bit) CRC
  setCRCLength( RF24_CRC_16 ) ;

  // Disable dynamic payloads, to match dynamic_payloads_enabled setting
  write_register(DYNPD,0);

  // Reset current status
  // Notice reset and flush is the last thing we do
  write_register(STATUS,1<<(RX_DR) | 1<<(TX_DS) | 1<<(MAX_RT) );

  // Set up default configuration.  Callers can always change it later.
  // This channel should be universally safe and not bleed over into adjacent
  // spectrum.
  setChannel(76);

  // Flush buffers
  flush_rx();
  flush_tx();
}
```

```
/****************************************************************************/

void RF24::startListening(void)
{
  write_register(CONFIG, read_register(CONFIG) | 1<<(PWR_UP) | 1<<(PRIM_RX));
  write_register(STATUS, 1<<(RX_DR) | 1<<(TX_DS) | 1<<(MAX_RT) );

  // Restore the pipe0 adddress, if exists
  if (pipe0_reading_address)
    write_register(RX_ADDR_P0, reinterpret_cast<const uint8_t*>(&pipe0_reading_address), 5);

  // Flush buffers
  flush_rx();
  flush_tx();

  // Go!
  ce(HIGH);

  // wait for the radio to come up (130us actually only needed)
  delayMicroseconds(130);
}

/****************************************************************************/

void RF24::stopListening(void)
{
  ce(LOW);
  flush_tx();
  flush_rx();
}

/****************************************************************************/

void RF24::powerDown(void)
{
  write_register(CONFIG,read_register(CONFIG) & ~1<<(PWR_UP));
}

/****************************************************************************/

void RF24::powerUp(void)
{
  write_register(CONFIG,read_register(CONFIG) | 1<<(PWR_UP));
}

/****************************************************************/

bool RF24::write( const void* buf, uint8_t len )
{
  bool result = false;
```

```
// Begin the write
startWrite(buf,len);

// ------------
// At this point we could return from a non-blocking write, and then call
// the rest after an interrupt

// Instead, we are going to block here until we get TX_DS (transmission completed and ack'd)
// or MAX_RT (maximum retries, transmission failed).  Also, we'll timeout in case the radio
// is flaky and we get neither.

// IN the end, the send should be blocking.  It comes back in 60ms worst case, or much faster
// if I tighted up the retry logic.  (Default settings will be 1500us.
// Monitor the send
uint8_t observe_tx;
uint8_t status;
uint32_t sent_at = millis();
const uint32_t timeout = 500; //ms to wait for timeout
do
{
  status = read_register(OBSERVE_TX,&observe_tx,1);
  IF_SERIAL_DEBUG(Serial.print(observe_tx,HEX));
}
while( ! ( status & ( 1<<(TX_DS) | 1<<(MAX_RT) ) ) && ( millis() - sent_at < timeout ) );

// The part above is what you could recreate with your own interrupt handler,
// and then call this when you got an interrupt
// ------------

// Call this when you get an interrupt
// The status tells us three things
// * The send was successful (TX_DS)
// * The send failed, too many retries (MAX_RT)
// * There is an ack packet waiting (RX_DR)
bool tx_ok, tx_fail;
whatHappened(tx_ok,tx_fail,ack_payload_available);

//printf("%u%u%u\r\n",tx_ok,tx_fail,ack_payload_available);

result = tx_ok;
IF_SERIAL_DEBUG(Serial.print(result?"...OK.":"...Failed"));

// Handle the ack packet
if ( ack_payload_available )
{
  ack_payload_length = getDynamicPayloadSize();
  IF_SERIAL_DEBUG(Serial.print("[AckPacket]/"));
  IF_SERIAL_DEBUG(Serial.println(ack_payload_length,DEC));
}
```

```
  // Yay, we are done.

  // Power down
  powerDown();

  // Flush buffers (Is this a relic of past experimentation, and not needed anymore??)
  flush_tx();

  return result;
}
/****************************************************************************/

void RF24::startWrite( const void* buf, uint8_t len )
{
  // Transmitter power-up
  write_register(CONFIG, ( read_register(CONFIG) | 1<<(PWR_UP) ) & ~1<<(PRIM_RX) );
  delayMicroseconds(150);

  // Send the payload
  write_payload( buf, len );

  // Allons!
  ce(HIGH);
  delayMicroseconds(15);
  ce(LOW);
}

/****************************************************************************/

uint8_t RF24::getDynamicPayloadSize(void)
{
  uint8_t result = 0;

  csn(LOW);
  SPI.transfer(52, R_RX_PL_WID );
  result = SPI.transfer(52, 0xff);
  csn(HIGH);

  return result;
}

/****************************************************************************/

bool RF24::available(void)
{
  return available(NULL);
}

/****************************************************************************/

bool RF24::available(uint8_t* pipe_num)
```

```
{
  uint8_t status = get_status();

  // Too noisy, enable if you really want lots o data!!
  //IF_SERIAL_DEBUG(print_status(status));

  bool result = ( status & 1<<(RX_DR) );

  if (result)
  {
    // If the caller wants the pipe number, include that
    if ( pipe_num )
      *pipe_num = ( status >> RX_P_NO ) & B111;

    // Clear the status bit

    // ??? Should this REALLY be cleared now?  Or wait until we
    // actually READ the payload?

    write_register(STATUS,1<<(RX_DR) );

    // Handle ack payload receipt
    if ( status & 1<<(TX_DS) )
    {
      write_register(STATUS,1<<(TX_DS));
    }
  }

  return result;
}

/****************************************************************************/

bool RF24::read( void* buf, uint8_t len )
{
  // Fetch the payload
  read_payload( buf, len );

  // was this the last of the data available?
  return read_register(FIFO_STATUS) & 1<<(RX_EMPTY);
}

/****************************************************************************/

void RF24::whatHappened(bool& tx_ok,bool& tx_fail,bool& rx_ready)
{
  // Read the status & reset the status in one easy call
  // Or is that such a good idea?
  uint8_t status = write_register(STATUS,1<<(RX_DR) | 1<<(TX_DS) | 1<<(MAX_RT) );

  // Report to the user what happened
```

```
  tx_ok = status & 1<<(TX_DS);
  tx_fail = status & 1<<(MAX_RT);
  rx_ready = status & 1<<(RX_DR);
}

/****************************************************************************/

void RF24::openWritingPipe(uint64_t value)
{
  // Note that AVR 8-bit uC's store this LSB first, and the NRF24L01(+)
  // expects it LSB first too, so we're good.

  write_register(RX_ADDR_P0, reinterpret_cast<uint8_t*>(&value), 5);
  write_register(TX_ADDR, reinterpret_cast<uint8_t*>(&value), 5);

  const uint8_t max_payload_size = 32;
  write_register(RX_PW_P0,min(payload_size,max_payload_size));
}

/****************************************************************************/

static const uint8_t child_pipe[] PROGMEM =
{
  RX_ADDR_P0, RX_ADDR_P1, RX_ADDR_P2, RX_ADDR_P3, RX_ADDR_P4, RX_ADDR_P5
};
static const uint8_t child_payload_size[] PROGMEM =
{
  RX_PW_P0, RX_PW_P1, RX_PW_P2, RX_PW_P3, RX_PW_P4, RX_PW_P5
};
static const uint8_t child_pipe_enable[] PROGMEM =
{
  ERX_P0, ERX_P1, ERX_P2, ERX_P3, ERX_P4, ERX_P5
};

void RF24::openReadingPipe(uint8_t child, uint64_t address)
{
  // If this is pipe 0, cache the address.  This is needed because
  // openWritingPipe() will overwrite the pipe 0 address, so
  // startListening() will have to restore it.
  if (child == 0)
    pipe0_reading_address = address;

  if (child <= 6)
  {
    // For pipes 2-5, only write the LSB
    if ( child < 2 )
      write_register(pgm_read_byte(&child_pipe[child]), reinterpret_cast<const uint8_t*>(&address), 5);
    else
      write_register(pgm_read_byte(&child_pipe[child]), reinterpret_cast<const uint8_t*>(&address), 1);

    write_register(pgm_read_byte(&child_payload_size[child]),payload_size);
```

```
    // Note it would be more efficient to set all of the bits for all open
    // pipes at once.  However, I thought it would make the calling code
    // more simple to do it this way.
    write_register(EN_RXADDR,read_register(EN_RXADDR) | 1<<(pgm_read_byte(&child_pipe_enable[child])));
  }
}

/****************************************************************************/

void RF24::toggle_features(void)
{
  csn(LOW);
  SPI.transfer(52, ACTIVATE );
  SPI.transfer(52, 0x73 );
  csn(HIGH);
}

/****************************************************************************/

void RF24::enableDynamicPayloads(void)
{
  // Enable dynamic payload throughout the system
  write_register(FEATURE,read_register(FEATURE) | 1<<(EN_DPL) );

  // If it didn't work, the features are not enabled
  if ( ! read_register(FEATURE) )
  {
    // So enable them and try again
    toggle_features();
    write_register(FEATURE,read_register(FEATURE) | 1<<(EN_DPL) );
  }

  IF_SERIAL_DEBUG(printf("FEATURE=%i\r\n",read_register(FEATURE)));

  // Enable dynamic payload on all pipes
  //
  // Not sure the use case of only having dynamic payload on certain
  // pipes, so the library does not support it.
  write_register(DYNPD,read_register(DYNPD) | 1<<(DPL_P5) | 1<<(DPL_P4) | 1<<(DPL_P3) | 1<<(DPL_P2) | 1<<(DPL_P1) | 1<<(DPL_P0));

  dynamic_payloads_enabled = true;
}

/****************************************************************************/

void RF24::enableAckPayload(void)
{
  //
  // enable ack payload and dynamic payload features
  //
```

```
  write_register(FEATURE,read_register(FEATURE) | 1<<(EN_ACK_PAY) | 1<<(EN_DPL) );

  // If it didn't work, the features are not enabled
  if ( ! read_register(FEATURE) )
  {
    // So enable them and try again
    toggle_features();
    write_register(FEATURE,read_register(FEATURE) | 1<<(EN_ACK_PAY) | 1<<(EN_DPL) );
  }

  IF_SERIAL_DEBUG(printf("FEATURE=%i\r\n",read_register(FEATURE)));

  //
  // Enable dynamic payload on pipes 0 & 1
  //

  write_register(DYNPD,read_register(DYNPD) | 1<<(DPL_P1) | 1<<(DPL_P0));
}

/****************************************************************************/

void RF24::writeAckPayload(uint8_t pipe, const void* buf, uint8_t len)
{
  const uint8_t* current = reinterpret_cast<const uint8_t*>(buf);

  csn(LOW);
  SPI.transfer(52, W_ACK_PAYLOAD | ( pipe & B111 ) );
  const uint8_t max_payload_size = 32;
  uint8_t data_len = min(len,max_payload_size);
  while ( data_len-- )
    SPI.transfer(52, *current++);

  csn(HIGH);
}

/****************************************************************************/

bool RF24::isAckPayloadAvailable(void)
{
  bool result = ack_payload_available;
  ack_payload_available = false;
  return result;
}

/****************************************************************************/

bool RF24::isPVariant(void)
{
  return p_variant ;
}
```

```
/****************************************************************************/

void RF24::setAutoAck(bool enable)
{
  if ( enable )
    write_register(EN_AA, B111111);
  else
    write_register(EN_AA, 0);
}

/****************************************************************************/

void RF24::setAutoAck( uint8_t pipe, bool enable )
{
  if ( pipe <= 6 )
  {
    uint8_t en_aa = read_register( EN_AA ) ;
    if( enable )
    {
      en_aa |= 1<<(pipe) ;
    }
    else
    {
      en_aa &= ~1<<(pipe) ;
    }
    write_register( EN_AA, en_aa ) ;
  }
}

/****************************************************************************/

bool RF24::testCarrier(void)
{
  return ( read_register(CD) & 1 );
}

/****************************************************************************/

bool RF24::testRPD(void)
{
  return ( read_register(RPD) & 1 ) ;
}

/****************************************************************************/

void RF24::setPALevel(rf24_pa_dbm_e level)
{
  uint8_t setup = read_register(RF_SETUP) ;
  setup &= ~(1<<(RF_PWR_LOW) | 1<<(RF_PWR_HIGH)) ;
```

```
  // switch uses RAM (evil!)
  if ( level == RF24_PA_MAX )
  {
    setup |= (1<<(RF_PWR_LOW) | 1<<(RF_PWR_HIGH)) ;
  }
  else if ( level == RF24_PA_HIGH )
  {
    setup |= 1<<(RF_PWR_HIGH) ;
  }
  else if ( level == RF24_PA_LOW )
  {
    setup |= 1<<(RF_PWR_LOW);
  }
  else if ( level == RF24_PA_MIN )
  {
    // nothing
  }
  else if ( level == RF24_PA_ERROR )
  {
    // On error, go to maximum PA
    setup |= (1<<(RF_PWR_LOW) | 1<<(RF_PWR_HIGH)) ;
  }

  write_register( RF_SETUP, setup ) ;
}

/****************************************************************************/

rf24_pa_dbm_e RF24::getPALevel(void)
{
  rf24_pa_dbm_e result = RF24_PA_ERROR ;
  uint8_t power = read_register(RF_SETUP) & (1<<(RF_PWR_LOW) | 1<<(RF_PWR_HIGH)) ;

  // switch uses RAM (evil!)
  if ( power == (1<<(RF_PWR_LOW) | 1<<(RF_PWR_HIGH)) )
  {
    result = RF24_PA_MAX ;
  }
  else if ( power == 1<<(RF_PWR_HIGH) )
  {
    result = RF24_PA_HIGH ;
  }
  else if ( power == 1<<(RF_PWR_LOW) )
  {
    result = RF24_PA_LOW ;
  }
  else
  {
    result = RF24_PA_MIN ;
  }
```

```
  return result ;
}

/****************************************************************************/

bool RF24::setDataRate(rf24_datarate_e speed)
{
  bool result = false;
  uint8_t setup = read_register(RF_SETUP) ;

  // HIGH and LOW '00' is 1Mbs - our default
  wide_band = false ;
  setup &= ~(1<<(RF_DR_LOW) | 1<<(RF_DR_HIGH)) ;
  if( speed == RF24_250KBPS )
  {
    // Must set the RF_DR_LOW to 1; RF_DR_HIGH (used to be RF_DR) is already 0
    // Making it '10'.
    wide_band = false ;
    setup |= 1<<( RF_DR_LOW ) ;
  }
  else
  {
    // Set 2Mbs, RF_DR (RF_DR_HIGH) is set 1
    // Making it '01'
    if ( speed == RF24_2MBPS )
    {
      wide_band = true ;
      setup |= 1<<(RF_DR_HIGH);
    }
    else
    {
      // 1Mbs
      wide_band = false ;
    }
  }
  write_register(RF_SETUP,setup);

  // Verify our result
  if ( read_register(RF_SETUP) == setup )
  {
    result = true;
  }
  else
  {
    wide_band = false;
  }

  return result;
}

/****************************************************************************/
```

```
rf24_datarate_e RF24::getDataRate( void )
{
  rf24_datarate_e result ;
  uint8_t dr = read_register(RF_SETUP) & (1<<(RF_DR_LOW) | 1<<(RF_DR_HIGH));

  // switch uses RAM (evil!)
  // Order matters in our case below
  if ( dr == 1<<(RF_DR_LOW) )
  {
    // '10' = 250KBPS
    result = RF24_250KBPS ;
  }
  else if ( dr == 1<<(RF_DR_HIGH) )
  {
    // '01' = 2MBPS
    result = RF24_2MBPS ;
  }
  else
  {
    // '00' = 1MBPS
    result = RF24_1MBPS ;
  }
  return result ;
}

/****************************************************************************/

void RF24::setCRCLength(rf24_crclength_e length)
{
  uint8_t config = read_register(CONFIG) & ~( 1<<(CRCO) | 1<<(EN_CRC)) ;

  // switch uses RAM (evil!)
  if ( length == RF24_CRC_DISABLED )
  {
    // Do nothing, we turned it off above.
  }
  else if ( length == RF24_CRC_8 )
  {
    config |= 1<<(EN_CRC);
  }
  else
  {
    config |= 1<<(EN_CRC);
    config |= 1<<( CRCO );
  }
  write_register( CONFIG, config ) ;
}

/****************************************************************************/
```

```
rf24_crclength_e RF24::getCRCLength(void)
{
  rf24_crclength_e result = RF24_CRC_DISABLED;
  uint8_t config = read_register(CONFIG) & ( 1<<(CRCO) | 1<<(EN_CRC)) ;

  if ( config & 1<<(EN_CRC ) )
  {
    if ( config & 1<<(CRCO) )
      result = RF24_CRC_16;
    else
      result = RF24_CRC_8;
  }

  return result;
}

/****************************************************************************/

void RF24::disableCRC( void )
{
  uint8_t disable = read_register(CONFIG) & ~1<<(EN_CRC) ;
  write_register( CONFIG, disable ) ;
}

/****************************************************************************/
void RF24::setRetries(uint8_t delay, uint8_t count)
{
 write_register(SETUP_RETR,(delay&0xf)<<ARD | (count&0xf)<<ARC);
}

// vim:ai:cin:sts=2 sw=2 ft=cpp
```