CrossWorks Examples Package

Version: 3.6

Contents

CrossWorks Examples Package

CrossWorks Examples Package

About the CrossWorks Examples Package

The *CrossWorks Examples Package* package delivers source code examples that are shared between many board support packages.

Delivery format

The *CrossWorks Examples Package* is delivered in source form.

Feedback

This facility is a work in progress and may undergo rapid change. If you have comments, observations, suggestions, or problems, please feel free to air them on the **CrossWorks Target and Platform API** discussion forum.

License

The following terms apply to the Rowley Associates Examples Package.

CrossWorks Examples Package

Crazyflie remote

About Crazyflie

The Crazyflie Nano is a small quadcopter that you can purchase from Bitcraze.

http://www.bitcraze.se/

It's great fun to fly!

The mess...

The problem I had with the Crazyflie is that if you want to fly it, you need to install the PC client on a laptop, tether a joystick, have Crazyradio sticking out from a USB port ready to be broken off, set the whole damn lot up, and then go fly.

It's not really a great solution, so I decided that I would write my own remote control software using CrossWorks and off-the-shelf hardware, along with the extensive support we've added in the **Platform Library**, the **Graphics Library**, and the **Device Library**.

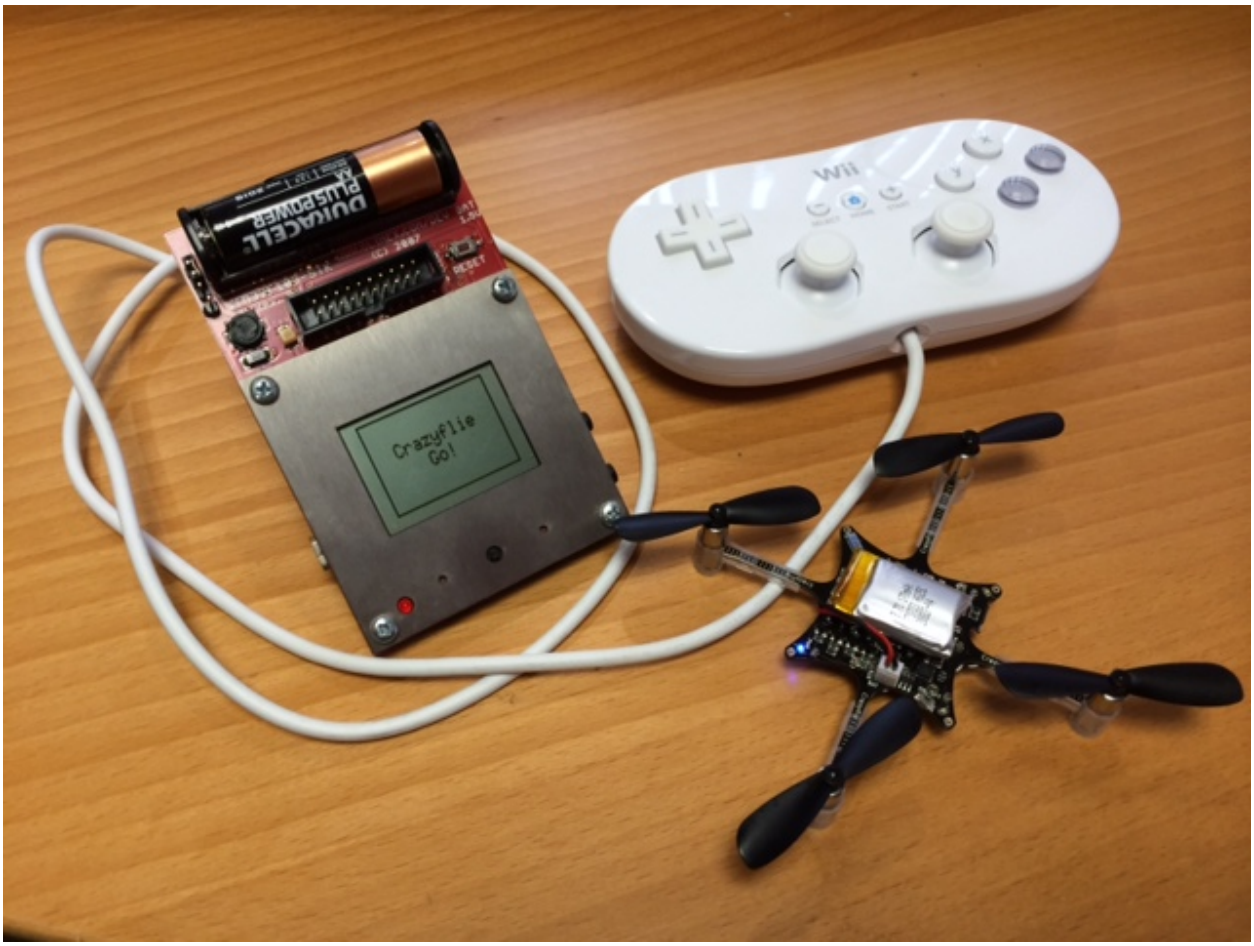
Core hardware

I chose an Olimex STM32-103STK as this has an nRF24L01 built in, the same type of device you need to communicate with the Crazyflie.

http://www.olimex.com/Products/ARM/ST/STM32-103STK/

What's more, it has a built-in LCD, is battery powered, and has a UEXT socket that you can connect a MOD-WII and a Wii Classic Controller to. In short, this is the *perfect* platform to build a no-nonsense stand-along Crazyflie remote control!

This is what I ended up with:



Olimex have also introduced the pretty-much identical STM32-405STK. This has an updated processor on it, but the increase in compute power doesn't bring anything to the remote control. It's a bit more expensive than the STM32-103STK, but clearly there is an insatiable demand for low-cost powerful processors! Needless to say, the Crazyflie remote software I wrote works on this board too.

http://www.olimex.com/Products/ARM/ST/STM32-405STK/

Setup

In the back of the STM32-103STK you'll need to plug a MOD-WII and a Nintendo Classic Controller. Install a battery and move the EXT/BAT jumper to power the device from the battery. Program the board with the firmware, and get ready to fly!

User interface

The LCD displays the current state of the software. After reset, devices are initialized and the software searches to find the channel to communicate with the Crazyflie. If it can't find it, just reset the board using the RESET button near the battery and try again. Once the Crazyflie is found, the red LED on the faceplate illuminates to show a good link—if the link drops, the LED goes out.

The left thumbstick controls thrust:

- Push up to increase thrust, center position is zero thrust.

The right thumbstick controls pitch and roll:

- Up-Down controls pitch
- Left-right controls roll.

The buttons control trim:

- X-B adjust pitch trim.
- Y-A adjust roll trim.

That's it! Up, up, and away: happy flying!

The code

```
/* Copyright (c) 2004-2013 Rowley Associates Limited.
*/

#include "libplatform/platform.h"
#include "libplatform/platform_graphics.h"
#include "libugfx/ctl_gfx.h"
#include "libdevice/wii_controller.h"
#include "example_support.h"
#include <cross_studio_io.h>
#include <string.h>

// This will compile "out of the box" on the Olimex STM32-405STK
// and Olimex STM32-103STK.

#if defined(Olimex_STM32_405STK)

#define NRF24L01_SS      STM32_405STK_NRF24L01_SS
#define NRF24L01_CE      STM32_405STK_NRF24L01_CE

#elif defined(Olimex_STM32_103STK)

#define NRF24L01_SS      STM32_103STK_NRF24L01_SS
#define NRF24L01_CE      STM32_103STK_NRF24L01_CE

#else

#error unupported platform

#endif

typedef struct {
  unsigned char c;
  float roll;
  float pitch;
  float yaw;
  unsigned short thrust;
} __attribute__((packed)) CRAZYFLIE_COMMANDER_SETPOINT_t;

typedef struct {
  unsigned char c;
  union {
```

```
    unsigned char data[32];  // one larger, to insert a null.
  } console;
} __attribute__((packed)) CRAZYFLIE_CONSOLE_t;

// NRF24 variants:
typedef enum {
  NRF24L01,     // nRF24L01
  NRF24L01P,    // nRF24L01+
} NRF24_VARIANT_t;


// NRF24 registers
typedef enum {
  NRF24_CONFIG       = 0x00,
  NRF24_EN_AA,
  NRF24_EN_RXADDR,
  NRF24_SETUP_AW,
  NRF24_SETUP_RETR,
  NRF24_RF_CH,
  NRF24_RF_SETUP,
  NRF24_STATUS,
  NRF24_OBSERVE_TX,  // 0x08
  NRF24_CD,
  NRF24_RX_ADDR_P0,
  NRF24_RX_ADDR_P1,
  NRF24_RX_ADDR_P2,
  NRF24_RX_ADDR_P3,
  NRF24_RX_ADDR_P4,
  NRF24_RX_ADDR_P5,
  NRF24_TX_ADDR,     // 0x10
  NRF24_RX_PW_P0,
  NRF24_RX_PW_P1,
  NRF24_RX_PW_P2,
  NRF24_RX_PW_P3,
  NRF24_RX_PW_P4,
  NRF24_RX_PW_P5,
  NRF24_FIFO_STATUS,
  NRF24_DYNPD        = 0x1c,
  NRF24_FEATURE
} NRF24_REGISTER_t;

// NRF24 CONFIG register bits.
#define NRF24_CONFIG_MASK_RX_DR      0x40
#define NRF24_CONFIG_MASK_TX_DS      0x20     // Asserted when ACK packet received
#define NRF24_CONFIG_MASK_MAX_RT     0x10
#define NRF24_CONFIG_EN_CRC          0x08
#define NRF24_CONFIG_CRCO            0x04
#define NRF24_CONFIG_PWR_UP          0x02
#define NRF24_CONFIG_PRIM_RX         0x01

// NRF24 EN_AA register bits.
#define NRF24_EN_AA_ENAA_P5          0x20
#define NRF24_EN_AA_ENAA_P4          0x10
#define NRF24_EN_AA_ENAA_P3          0x08
#define NRF24_EN_AA_ENAA_P2          0x04
#define NRF24_EN_AA_ENAA_P1          0x02
#define NRF24_EN_AA_ENAA_P0          0x01

// NRF24 EN_AA register bits.
#define NRF24_EN_RXADDR_ERX_P5       0x20
#define NRF24_EN_RXADDR_ERX_P4       0x10
#define NRF24_EN_RXADDR_ERX_P3       0x08
```

```
#define NRF24_EN_RXADDR_ERX_P2       0x04
#define NRF24_EN_RXADDR_ERX_P1       0x02
#define NRF24_EN_RXADDR_ERX_P0       0x08

// NRF24 RF_SETUP bits
#define NRF24_RF_SETUP_CONT_WAVE     0x80     // NRF24L01+ only: continuous carrier
#define NRF24_RF_SETUP_RF_DR_LOW     0x20     // low data rate,
 250 kHz, need RF_DR_HIGH=0 if set
#define NRF24_RF_SETUP_PLL_LOCK      0x10
#define NRF24_RF_SETUP_RF_DR_HIGH    0x08
#define NRF24_RF_SETUP_RF_PWR        0x02
#define NRF24_RF_SETUP_RF_LNA_HCURR  0x01

// NRF24 data rates
#define NRF24_RF_SETUP_RF_DR_250kbps (1*NRF24_RF_SETUP_RF_DR_LOW + 0*NRF24_RF_SETUP_RF_DR_HIGH)
#define NRF24_RF_SETUP_RF_DR_1Mbps
  (0*NRF24_RF_SETUP_RF_DR_LOW + 0*NRF24_RF_SETUP_RF_DR_HIGH)
#define NRF24_RF_SETUP_RF_DR_2Mbps
  (0*NRF24_RF_SETUP_RF_DR_LOW + 1*NRF24_RF_SETUP_RF_DR_HIGH)
#define NRF24_RF_SETUP_RF_DR_MASK
  (1*NRF24_RF_SETUP_RF_DR_LOW + 1*NRF24_RF_SETUP_RF_DR_HIGH)

// NRF24 STATUS bits
#define NRF24_STATUS_RX_DR           0x40     // Rx data ready interrupt
#define NRF24_STATUS_TX_DS           0x20     // Tx data sent interrupt
#define NRF24_STATUS_MAX_RT          0x10     // Maximum number of transmit interrupt

typedef enum {
  NRF24_CMD_R_REGISTER          = 0x00,
  NRF24_CMD_W_REGISTER          = 0x20,
  NRF24_CMD_RX_PAYLOAD          = 0x61,
  NRF24_CMD_TX_PAYLOAD          = 0xA0,
  NRF24_CMD_FLUSH_TX            = 0xE1,
  NRF24_CMD_FLUSH_RX            = 0xE2,
  NRF24_CMD_TX_PL               = 0xE3,
  NRF24_CMD_ACTIVATE            = 0x50,
  NRF24_CMD_R_RX_PL_WID         = 0x60,
  NRF24_CMD_W_ACK_PAYLOAD       = 0xA8,
  NRF24_CMD_W_TX_PAYLOAD_NO_ACK = 0xB0,
  NRF24_CMD_NOP                 = 0xFF
} NRF24_CMD_t;

typedef struct NRF24_RADIO_s {
  CTL_SPI_DEVICE_t dev;
  void (*write_ce)(struct NRF24_RADIO_s *, int state);

  // Shadow registers.
  unsigned short setup;
  unsigned short retr;

  // Operating mode.
  NRF24_VARIANT_t variant;
  unsigned short ack_payload_length;
} NRF24_RADIO_t;

static const PLATFORM_UEXT_CONFIGURATION_t *uext;
static float trim_pitch;
static float trim_roll;

static void
nrf24_write_register(NRF24_RADIO_t *self, int reg, const void *addr, size_t len)
{
```

```
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_W_REGISTER | reg);
  ctl_spi_write(&self->dev, addr, len);
  ctl_spi_deselect_device(&self->dev);
}

static void
nrf24_read_register(NRF24_RADIO_t *self, int reg, void *addr, size_t len)
{
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_R_REGISTER | reg);
  ctl_spi_read(&self->dev, addr, len);
  ctl_spi_deselect_device(&self->dev);
}

static void
nrf24_write_8b_register(NRF24_RADIO_t *self, int reg, int value)
{
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_W_REGISTER | reg);
  ctl_spi_put(&self->dev, value);
  ctl_spi_deselect_device(&self->dev);
}

static CTL_STATUS_t
nrf24_read_8b_register(NRF24_RADIO_t *self, int reg)
{
  CTL_STATUS_t stat;
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_R_REGISTER | reg);
  stat = ctl_spi_get(&self->dev);
  ctl_spi_deselect_device(&self->dev);
  return stat;
}

static void
uext_nrf24_write_cs(CTL_SPI_DEVICE_t *dev, int state)
{
  platform_write_digital_pin(uext->pin10_ssel, state);
}

static void
uext_nrf24_write_ce(NRF24_RADIO_t *self, int state)
{
  platform_write_digital_pin(uext->pin4_rxd, state);
}

static void
builtin_nrf24_write_cs(CTL_SPI_DEVICE_t *dev, int state)
{
  platform_write_digital_pin(NRF24L01_SS, state);
}

static void
builtin_nrf24_write_ce(NRF24_RADIO_t *self, int state)
{
  platform_write_digital_pin(NRF24L01_CE, state);
}

static CTL_STATUS_t
nrf24_cmd(NRF24_RADIO_t *self, unsigned char cmd)
{
```

```
  unsigned char status;
  ctl_spi_select_device(&self->dev);
  ctl_spi_exchange(&self->dev, &cmd, &status, 1);
  ctl_spi_deselect_device(&self->dev);
  return status;
}

static CTL_STATUS_t
nrf24_nop(NRF24_RADIO_t *self)
{
  return nrf24_cmd(self, NRF24_CMD_NOP);
}

static CTL_STATUS_t
nrf24_flush_tx(NRF24_RADIO_t *self)
{
  return nrf24_cmd(self, NRF24_CMD_FLUSH_TX);
}

static CTL_STATUS_t
nrf24_flush_rx(NRF24_RADIO_t *self)
{
  return nrf24_cmd(self, NRF24_CMD_FLUSH_RX);
}

static CTL_STATUS_t
nrf24_set_channel(NRF24_RADIO_t *self, int channel)
{
  // Validate channel.
  if (channel < 0 || 125 < channel)
    return CTL_PARAMETER_ERROR;

  // Change the channel.
//  self->write_ce(self, 0);
  nrf24_write_8b_register(self, NRF24_RF_CH, channel);
//  self->write_ce(self, 1);

  // All done.
  return CTL_NO_ERROR;
}

static void
nrf24_recompute_retr(NRF24_RADIO_t *self)
{
  int ard = 0;

  // Retry timeouts depend upon link speed and payload length.
  switch (self->setup & NRF24_RF_SETUP_RF_DR_MASK)
    {
    case NRF24_RF_SETUP_RF_DR_2Mbps:
      if (self->ack_payload_length <= 15)      ard = 0;  // 250 us
      else                                     ard = 1;  // 500 us
      break;

    case NRF24_RF_SETUP_RF_DR_1Mbps:
      if (self->ack_payload_length <= 5)       ard = 0;  // 250 us
      else                                     ard = 1;  // 500 us
      break;

    case NRF24_RF_SETUP_RF_DR_250kbps:  // table 18
      if      (self->ack_payload_length <=  0) ard = 1;  //  500 us
      else if (self->ack_payload_length <=  8) ard = 2;  //  750 us
```

```
      else if (self->ack_payload_length <= 16) ard = 3;  // 1000 us
      else if (self->ack_payload_length <= 24) ard = 4;  // 1250 us
      else                                     ard = 5;  // 1500 us

    default:
      break;
    }

  // Integrate ARD into shadow register.
  self->retr &= ~(0xf << 4);
  self->retr |=  (ard << 4);

  // Synchronize hardware and shadow registers.
  nrf24_write_8b_register(self, NRF24_SETUP_RETR, self->retr);
}

static CTL_STATUS_t
nrf24_set_data_rate(NRF24_RADIO_t *self, int kHz)
{
  // Clear data rate bits in shadow register.
  self->setup &= ~NRF24_RF_SETUP_RF_DR_MASK;

  // nRF24L01 does not support 250 kHz low data rate...
  if (kHz <= 250 && self->variant == NRF24L01P)
    {
      self->setup |= NRF24_RF_SETUP_RF_DR_250kbps;\
    }
  else if (kHz <= 1000)
    {
      self->setup |= NRF24_RF_SETUP_RF_DR_1Mbps;
    }
  else
    {
      self->setup |= NRF24_RF_SETUP_RF_DR_2Mbps;
    }

  // Synchronize hardware and shadow registers.
  nrf24_write_8b_register(self, NRF24_RF_SETUP, self->setup);

  // Synchronize RETR.
  nrf24_recompute_retr(self);

  // All done.
  return CTL_NO_ERROR;
}

static CTL_STATUS_t
nrf24_set_auto_retransmit_count(NRF24_RADIO_t *self, int count)
{
  // Clamp.
  if      (count <  0) count = 0;
  else if (count > 15) count = 15;

  // Update.
  self->retr &= ~0xf;
  self->retr |= count;

  // Synchronize hardware and shadow registers.
  nrf24_write_8b_register(self, NRF24_SETUP_RETR, self->retr);

  // All done.
  return CTL_NO_ERROR;
```

```
}

static CTL_STATUS_t
nrf24_set_transmit_power(NRF24_RADIO_t *self, int dBm)
{
  // Clear power bits in shadow register.
  self->setup &= ~(3 * NRF24_RF_SETUP_RF_PWR);

  // Encode power in shadow register.
  if      (dBm <= -18) { self->setup |= 0*NRF24_RF_SETUP_RF_PWR; }
  else if (dBm <= -12) { self->setup |= 1*NRF24_RF_SETUP_RF_PWR; }
  else if (dBm <= -6)  { self->setup |= 2*NRF24_RF_SETUP_RF_PWR; }
  else                 { self->setup |= 3*NRF24_RF_SETUP_RF_PWR; }

  // Synchronize hardware and shadow registers.
  nrf24_write_8b_register(self, NRF24_RF_SETUP, self->setup);

  // All done.
  return CTL_NO_ERROR;
}

static CTL_STATUS_t
nrf24_set_ack_payload_length(NRF24_RADIO_t *self, int bytes)
{
  // Update ack payload length.
  self->ack_payload_length = bytes;

  // Update ARD based on payload length and data rate.
  nrf24_recompute_retr(self);
}

static void
nrf24_set_address(NRF24_RADIO_t *self, const unsigned char *address)
{
  int i;

  nrf24_write_register(self, NRF24_TX_ADDR, address, 5);
  for (i = 0; i < 5; ++i)
    nrf24_write_8b_register(self, NRF24_RX_ADDR_P0+i, address[i]);
}

static void
nrf24_pulse_ce(NRF24_RADIO_t *self)
{
  self->write_ce(self, 1);
  platform_spin_delay_us(20);  // From datasheet, must be > 10us
  self->write_ce(self, 0);
}

static void
nrf24_transmit_packet(NRF24_RADIO_t *self,
                      const void *payload, size_t payload_size)
{
  // Write payload to the buffer.
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_TX_PAYLOAD);
  ctl_spi_write(&self->dev, payload, payload_size);
  ctl_spi_deselect_device(&self->dev);

  // Pulse CE to start transmit.
  nrf24_pulse_ce(self);
}
```

```
static CTL_STATUS_t
nrf24_read_receive_packet(NRF24_RADIO_t *self, void *payload)
{
  CTL_STATUS_t stat;

  // Get the packet length.
  ctl_spi_select_device(&self->dev);
  ctl_spi_put(&self->dev, NRF24_CMD_R_RX_PL_WID);
  stat = ctl_spi_get(&self->dev);
  ctl_spi_deselect_device(&self->dev);

  // Validate response.
  if (0 < stat && stat <= 32)
    {
      ctl_spi_select_device(&self->dev);
      ctl_spi_put(&self->dev, NRF24_CMD_RX_PAYLOAD);
      ctl_spi_read(&self->dev, payload, stat);
      ctl_spi_deselect_device(&self->dev);
      return stat;
    }
  else
    return 0;
}

static CTL_STATUS_t
nrf24_wait_for_interrupt_and_acknowledge(NRF24_RADIO_t *self)
{
  unsigned interrupts;
  CTL_STATUS_t stat;

  // Interrupts to wait for.
  interrupts = NRF24_STATUS_RX_DR |
               NRF24_STATUS_TX_DS |
               NRF24_STATUS_MAX_RT;

  // FIXME: add timeout
  for (;;)
    {
      if ((stat = nrf24_nop(self)) & interrupts)
        break;
    }

  // Acknowledge interrupt.
  nrf24_write_8b_register(self, NRF24_STATUS, interrupts & stat);

  // Done.
  return stat;
}

static CTL_STATUS_t
nrf24_send_packet(NRF24_RADIO_t *self,
                  const void *payload, size_t payload_size,
                  void *ack, size_t *ack_size)
{
  CTL_STATUS_t stat;

  // Transmit packet.
  nrf24_transmit_packet(self, payload, payload_size);

  // Wait for interrupt.
  stat = nrf24_wait_for_interrupt_and_acknowledge(self);
```

```
  // Did we manage to send this packet?  If we took the maximum number of
  // transmits, just drop it.
  if (stat & NRF24_STATUS_MAX_RT)
    {
      nrf24_flush_tx(self);
      return CTL_DEVICE_NOT_RESPONDING;
    }

  // Did we receive an acknowledgment?  If so, accept it.
  if (stat & NRF24_STATUS_RX_DR)
    *ack_size = nrf24_read_receive_packet(self, ack);
  else
    *ack_size = 0;

  // Done with the acknowledge.
  nrf24_flush_rx(self);

  // Return "data sent" status.
  return stat & NRF24_STATUS_TX_DS;
}

static int crazyflie_channel;

static void
channel_scan(NRF24_RADIO_t *self)
{
  int separation, channel;
  CTL_STATUS_t stat;
  unsigned char ack[33];
  size_t ack_size;
  CRAZYFLIE_COMMANDER_SETPOINT_t setpoint;

  // Level, no thrust.
  setpoint.c = 3 << 4;
  setpoint.roll = 0;
  setpoint.pitch = 0;
  setpoint.yaw = 0;
  setpoint.thrust = 0;

  // Channel separation is different for 2Mbps link.
  separation = 1;
  if ((self->setup & NRF24_RF_SETUP_RF_DR_MASK) == NRF24_RF_SETUP_RF_DR_2Mbps)
    separation = 2;

  // When scanning, don't use lots of retransmits.
  nrf24_set_auto_retransmit_count(self, 5);

  // Scan channels...
  for (channel = 0; channel <= 125; channel += separation)
    {
      nrf24_set_channel(self, channel);
      stat = nrf24_send_packet(self, &setpoint, sizeof(setpoint), ack, &ack_size);
      if (stat > 0)
        {
//          debug_printf("Response on channel %d\n", channel);
          crazyflie_channel = channel;
          break;
        }
    }
}
```

```
static void
app_show_title(const char *text)
{
  const char *space;

  // Select font.
  ctl_gfx_context.current_font = &fixed_6x10;

  // Find delimiter.
  space = strchr(text, ' ');

  // Clear down.
  ctl_gfx_set_pen_color(ctl_gfx_driver->default_background);
  ctl_gfx_fill_rectangle_wh(0, 0, ctl_gfx_screen_width(), ctl_gfx_screen_height());

  // Draw 1st message.
  ctl_gfx_set_text_color(ctl_gfx_driver->default_foreground);
  ctl_gfx_set_pen_color(ctl_gfx_driver->default_foreground);
  ctl_gfx_draw_stringn((ctl_gfx_screen_width() - ctl_gfx_context.current_font-
>width * (space-text))/2,
                       ctl_gfx_screen_height()/2 - ctl_gfx_context.current_font->height,
                       text,
                       space-text);

  // Draw 2nd message.
  ctl_gfx_draw_string((ctl_gfx_screen_width() - ctl_gfx_context.current_font-
>width * strlen(space+1))/2,
                      ctl_gfx_screen_height()/2,
                      space+1);

  // Frame.
  ctl_gfx_draw_rectangle_wh(0, 0,
                            ctl_gfx_screen_width(), ctl_gfx_screen_height());
}

int
main(void)
{
  int i;
  CTL_STATUS_t stat;
  CTL_SPI_BUS_t *spi;
  CTL_I2C_BUS_t *bus;
  NRF24_RADIO_t radio;
  char data[5];
  unsigned old_press, new_press;

  // Initialize platform.
  platform_initialize();

  // Configure the built-in LCD.
  example_check_status(platform_configure_builtin_graphics());

  // Turn link LED off.
  platform_write_led(0, 0);

  // Show title and wait for user to read it.
  app_show_title("Crazyflie Remote");
  ctl_delay(2000);

  app_show_title("Initialize Controller");
  ctl_delay(1000);
```

```
  // Get UEXT descriptor.
  uext = platform_uext_configuration(0);

  // Configure platform pins for UEXT I2C.
  example_check_status(platform_configure_i2c_bus(uext->i2c_bus_index));

  // Get the I2C bus that is routed to the UEXT connector.
  bus = platform_i2c_bus(uext->i2c_bus_index);

  // Initialize sensor.
  example_check_status(wii_extension_controller_initialize(bus));

  app_show_title("Initialize Radio");
  ctl_delay(1000);

#if 0
  // This code is for a MOD-nRF24L01 on UEXT #0.

  // Get UEXT configuration.
  uext = platform_uext_configuration(0);

  // Configure platform pins for SPI on UEXT socket.
  example_check_status(platform_configure_spi_bus(uext->spi_bus_index));

  // Configure UEXT CS and CE.
  example_check_status(platform_configure_pin(uext-
>pin10_ssel, PIN_FUNCTION_DIGITAL_OUTPUT | PIN_CLAIM_EXCLUSIVE));  // CS
  example_check_status(platform_configure_pin(uext->pin4_rxd,
  PIN_FUNCTION_DIGITAL_OUTPUT | PIN_CLAIM_EXCLUSIVE));  // CE
  platform_write_digital_pin(uext->pin10_ssel, 1);
  platform_write_digital_pin(uext->pin4_rxd, 1);

  // Get the SPI bus routed to the UEXT socket.
  spi = platform_spi_bus(uext->spi_bus_index);

  // Initialize SPI device associated with NRF24L01 and attach it to the bus.
  memset(&dev, 0, sizeof(dev));
  dev.select = uext_write_cs;
  ctl_spi_attach_device(spi, &dev);

#else

  // STM32-103STK has nRF2401 on internal SPI bus.
  example_check_status(platform_configure_spi_bus(1, 0));

  // Configure UEXT CS and CE.
  example_check_status(platform_claim_pin(NRF24L01_SS, PIN_FUNCTION_DIGITAL_OUTPUT | PIN_CLAIM_EXCLUSIVE));
  // CS
  example_check_status(platform_claim_pin(NRF24L01_CE, PIN_FUNCTION_DIGITAL_OUTPUT | PIN_CLAIM_EXCLUSIVE));
  // CE
  platform_write_digital_pin(NRF24L01_SS, 1);
  platform_write_digital_pin(NRF24L01_CE, 0);

  // Get the SPI bus routed to the UEXT socket.
  spi = platform_spi_bus(1);

  // Initialize SPI device associated with NRF24L01 and attach it to the bus.
  memset(&radio, 0, sizeof(radio));
  radio.variant = NRF24L01;
  radio.dev.select = builtin_nrf24_write_cs;
  radio.write_ce = builtin_nrf24_write_ce;
  ctl_spi_attach_device(spi, &radio.dev);
```

```
#endif

  // See if we can read a register.
  ctl_spi_set_protocol(&radio.dev, CTL_SPI_MODE1, 8, 1000000, 0xff);

  // Select device.
  nrf24_read_register(&radio, 0x0a, data, 5);

  // Start radio in PTX mode.  Clear down interrupts.
  nrf24_write_8b_register(&radio, NRF24_CONFIG, NRF24_CONFIG_MASK_RX_DR |
                                                NRF24_CONFIG_MASK_TX_DS |
                                                NRF24_CONFIG_MASK_MAX_RT |
                                                NRF24_CONFIG_EN_CRC |
                                                NRF24_CONFIG_CRCO |
                                                NRF24_CONFIG_PWR_UP);

  // Wait for radio up.
  platform_spin_delay_ms(1);

  // Enable dynamic packet size and ACK payload features.
  nrf24_write_8b_register(&radio, NRF24_FEATURE, 0x06);
  nrf24_write_8b_register(&radio, NRF24_DYNPD, 0x01);

  // Configure fast data rate, high power, 32-byte ACK payload.
  nrf24_set_data_rate(&radio, 2000);
  nrf24_set_transmit_power(&radio, 0);
  nrf24_set_ack_payload_length(&radio, 32);
  nrf24_set_auto_retransmit_count(&radio, 10);

  // Zap any outstanding packets.
  for (i = 0; i < 10; ++i)
    {
      nrf24_flush_tx(&radio);
      nrf24_flush_rx(&radio);
    }

  // Show title and wait for user to read it.
  app_show_title("Scanning... Please wait");
  ctl_delay(1000);

  // Channel scan.
  channel_scan(&radio);

  // If we didn't detect it, drop out.
  if (crazyflie_channel == 0)
    {
      app_show_title("Crazyflie Not Detected");
      ctl_delay(1000);
      example_finish();
    }

  // Control copter.
  app_show_title("Crazyflie Go!");
  ctl_delay(1000);

  // Turn link LED on.
  platform_write_led(0, 1);

  // Send command packets.
  nrf24_set_channel(&radio, crazyflie_channel);
```

```
  // Say no trim buttons pressed.
  old_press = 0;

  // Into the blue!
  for (;;)
    {
      CRAZYFLIE_COMMANDER_SETPOINT_t setpoint;
      CRAZYFLIE_CONSOLE_t ack;
      WII_CLASSIC_REPORT_t report;
      unsigned ack_size;
      static int last_stat;

      // Sample.
      stat = wii_classic_sample(bus, &report);

      // If there's an error reading the controller, ignore reading.
      if (stat < CTL_NO_ERROR)
        continue;

      // Figure out if buttons have been pressed.
      new_press = ~old_press & report.buttons_pressed;
      old_press = report.buttons_pressed;

      // Adjust any trim values.
      if (new_press & WII_BUTTON_B)
        trim_pitch -= 1;
      if (new_press & WII_BUTTON_X)
        trim_pitch += 1;
      if (new_press & WII_BUTTON_A)
        trim_roll += 1;
      if (new_press & WII_BUTTON_Y)
        trim_roll -= 1;

      // Setpoint command.
      setpoint.c = 3 << 4;
      setpoint.roll = 0;
      setpoint.pitch = 0;
      setpoint.yaw = 0;

      // Left thumbstick Y controls thrust.
      if (report.left_joystick_y < 0x1f)
        setpoint.thrust = 0;
      else
        setpoint.thrust = (report.left_joystick_y - 0x1f) * 1875;

      // Right thumbstick controls roll and pitch.
      setpoint.pitch = (float)(16 - report.right_joystick_y) / 16 * 50;
      setpoint.roll  = (float)(report.right_joystick_x - 16) / 16 * 50;

      // Apply trims.
      setpoint.pitch += trim_pitch;
      setpoint.roll  += trim_roll;

#if 0
      // ATTENTION -- this is just to see if we're making sense...
      debug_printf("thrust: %6.2f   pitch: %6.2f\n",
                   setpoint.thrust, setpoint.pitch);
#endif

      // Send control packet.
      stat = nrf24_send_packet(&radio,
                               &setpoint, sizeof(setpoint),
```

```
                                &ack, &ack_size);

      // Reflect status on link LED: on when good link, off when lost.
      platform_write_led(0, stat > 0);

      // Dump console data coming back.
      if (stat >= 0 && ack_size > 0)
        {
          if ((ack.c >> 4) == 0)
            {
              // Console data.
              char *p = strnchr(ack.console.data, '\n', ack_size-1);
              if (p)
                p[1] = 0;
              else
                ack.console.data[ack_size-1] = 0;

#if 0
              // ATTENTION -- enable this if you want to see the messages
              // returned by the Crazyflie, but you need to run this under
              // the debugger!
              debug_printf("%s", ack.console.data);
#endif
            }
        }
    }

  // Done.
  return example_finish();
}
```

Defender

About Defender

For execution on a SolderCore and a SolderCore Arcade Shield or SolderCore LCD Shield. I've even run this code on a Windows PC using Qt to do GUI heavy lifting.

Background

This code replicates, as accurately as I can make it, a Williams Defender unit. Defender was one of those games that was pretty awesome for its time.

Please don't complain about the coding style, don't ask how it works, just do not bother me. I send this code out into the world to fend for itself and for you to unravel any puzzles you find. You have a SolderCore, you have an Arcade Shield, you have CrossWorks, you have a debugger, so all is not lost.

Core hardware

This code is primarily intended to run on a SolderCore and a SolderCore Arcade Shield or a SolderCore LCD Shield. Best gameplay comes from using the Arcade Shield because the display is bigger and it is considerably faster.

You can also run this code, using a SolderCore Arcade Shield or SolderCore LCD Shield, on:

- an Olimex STM32-E407.
- an Olimex STM32-H407.
- a BugBlat Cortino.
- a Netduino Plus 2.
- an mbed-LPC1768 with an ELMICRO TestBed. (This platform is a bit of a challenge as the LPC1768 RAM is split into several regions, none big enough accommodate a complete frame buffer.)

And you can run this code using the integrated LCD of:

- an Olimex STM32-LCD.

...and perhaps this code is included in CrossStudio as an Easter Egg? :-)

Human interface

The code can either use a Defender Playboard with a standard joystick and arcade buttons or a Nintendo Wii Classic Controller.

You can attach a Classic Controller using a SolderCore SenseCore and WiiChuk adapter. I happen to lay everything out using a 2x2 "flat four" base.

I've also coded up an interface using the Nintendo Wii Nunchuk Controller for the STM32-LCD in case you purchased one of those from Olimex. In this case, use the analog joystick for ship control and C to let off a smart bomb and Z to fire.

I laser-cut my Defender Playboard from 5mm acrylic and fitted some proper arcade buttons and a nice joystick. A good place to purchase these in the UK is **Gremlin Solutions**. You will find that 3mm acrylic is a better fit for the arcade buttons from Gremlin because they have a spring-latch underneath that will not lock on 5mm acrylic—I rebated the cuts for the buttons so mine would.

A warning: I purchased some arcade buttons from SparkFun but these are very deep and have a seriously naff feel. Don't use these arcade buttons, they are truly awful.

What's different

I took a little artistic license with the gameplay:

- The two-player game pits you against an AI-controlled Defender that is on screen, and playing, when you play. Neither Defender can collide with the other Defender, and neither Defender can shoot down or smart bomb the other Defender.
- The game doesn't stop and restart the wave when you die. This is a consequence of two-player mode. Although it would be possible to restart, I quite like it this way.

What's not implemented

Some things have not been implemented yet, and I may well get round to implementing them when I feel the need. Things left out for the moment are:

- Sound effects. I started putting in the hooks for sound effects, but I am no sound designer and haven't found a satisfactory way to get sound effects integrated into a SolderCore setup. Ideas run along the lines of the GinSing, the Fluxamasynth (MIDI), or custom VS1053 firmware (don't want to do this…), and then dry up.
- Baiter hurry-ups.
- Exploding landscape and hyperspace on loss of last human.
- Warping when pressing the hyperspace button.
- High scores. Who needs 'em? :-)
- The AI could be much better. Have a go!

Minimal FTP Server

Minimal FTP Server README

This note is a description of the FTP server example.

Overview

The FTP server example is a minimal implementation of an FTP server. It will serve simultaneous client connections to the server if you configure `MAXIMUM_FTP_CLIENTS` in `example_ftp_server.c`.

Limitations

The server is *minimal* and therefore has certain limitations. If all you wish to do is store and retrieve files from an SD card managed by the Mass Storage Library, this will do that for you. It will not, however, rename files or act as a full FTP server: that is not its purpose.

This server has no compile-time configuration options. If you wish to remove PUT or GET capability, do this by editing the code. You can extend the capabilities of the server, and customize it for your needs, as it is delivered in source form.

Minimal HTTP Server

Minimal HTTP Server README

This note is a description of the HTTP server example.

Overview

The HTTP server example is a minimal implementation of an HTTP server that serves pages from a mounted disk. It will serve simultaneous client connections to the server if you configure `MAXIMUM_HTTP_CLIENTS` in `ctl_http_server.c`.

Limitations

The server is *minimal* and therefore has certain limitations. If all you wish to do is serve files from an SD card managed by the Mass Storage Library, this will do that for you. It will not, however, provide capabilities such as POST, CGI, and so on.

This server has no compile-time configuration options. If you wish to remove capabilities, do this by editing the code. You can extend the capabilities of the server, and customize it for your needs, as it is delivered in source form.

Weather Station LCD1x9

About the example

The application searches an I2C bus to find a light sensor, a pressure sensor, a humidity sensor, and a temperature sensor. After enumerating the available sensors, it will show a carousel of measurements on the LCD.

MOD-LCD1x9 Setup

For boards with a UEXT socket:

- Plug a MOD-LCD1x9 into the first UEXT socket.

On everything else:

- Wire the MOD-LCD1x9 SDA/SCL signals to the primary platform I2C bus.

Arduino-format setup

You can use a SenseCore with a CorePressure module and CoreLight module, for instance. Or you can use a Jee Labs plug shield with some sensors that they offer. Or you can plug both of them in at once, if you really want to.