PulseOn OHR Module Application Note

May 10 2015

Version 0.95

15 Apr 2016

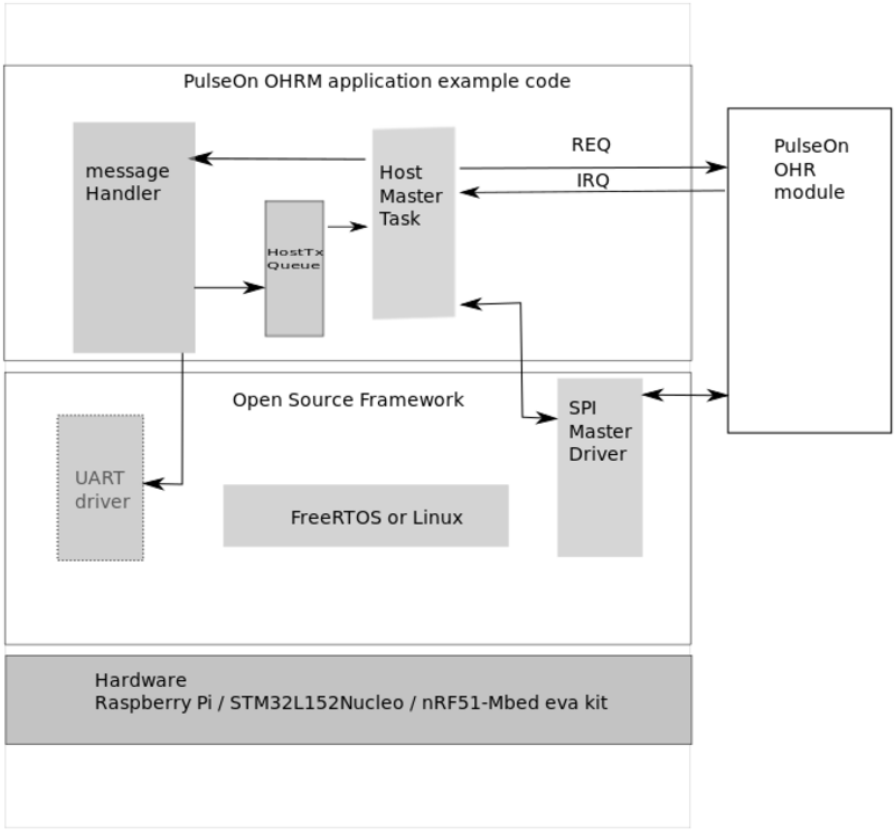
Table of Contents

1 Introduction

The PulseOn OHRM application example demonstrates how to use OHRM. It is based on OpenSource Frameworks that could be used as part of customer closed source application.

First application examples can run on Raspberry Pi board or on Nordic semiconductor nRF51 Mbed evaluation board or on ST STM32L152 Nucleo board.

2 Structure



Example host application block diagram

2.0.1 Open Source Framework

Open Source Framework contains operating system and required drivers. PulseOn supports Linux and FreeRTOS but code could be ported to other embedded RTOS. Framework contains no PulseOn trade secrets or proprietary code. All code could be used as part of customer application.

2.0.2 Raspberry Pi

The RaspberryPi application example uses a regular Raspbian Linux distribution. Raspbian provides SPI and Serial/Console port drivers. The WiringPi library is required to handle user mode interrupts.

2.0.3 STM32 and nRF51

STM32 and nRF51 example is based on the FreeRTOS real time operating system. Additional hardware dependent SPI and serial console drivers are supplied by PulseOn. The serial and SPI driver code is based on FreeRTOS, STM and Nordic semi-open code.

2.1 Framework components

2.1.1 SPI Master Driver

The SPI Master Driver is the only part required on top of the base FreeRTOS in a customer application. The SPI driver is responsible for transferring data frames from and to the module. The SPI Driver is interrupt-based and releases the processor to other tasks when data transfer is in progress.

2.1.2 UART Driver

The UART driver is only required for the console-output for the example application. The example application outputs the heart rate data it receives. The UART driver is interrupt-based, and it releases the CPU to other tasks when a transfer is in progress.

3 PulseOn OHRM application example code

The application example code contains two parts, hostTask and messageHandler. All application examples are PulseOn proprietary code that PulseOn has licensed to customers.

3.1 hostTask

The HostTask is responsible for implementing the communication with the module. When the module indicates (using the IRQ line) that data is available, it does uses SPI Transfer to receive data from module.

If there is a frame to send to the module, hostTask issues REQ line to initiate transfer. hostTask does not need to know the content of frames sent to or received from the module. When the hostTask receives frames from the module, it calls the messageHandler. HostTask implements a send message queue, hostTxQueue. When the customer application would like to either change usage mode of module or set certain configurations on the module, it should queue message frames to hostTxQueue. The host task then issues the REQ line and transfers the message to the module.

3.1.1 messageHandler

The Message handler represents customer application code. It receives message frames from OHRM via hostTask. Our example messageHandler just prints received messages to the serial port.

3.1.2 Software Updater

The Software Updater is responsible for updating the module firmware.

TBD

4 Module Protocol

This section describes how the communication works between host and the module with different protocols. The protocol is selected during module start-up, for details please see Device startup section.

4.1 SPI

The SPI interface has two lines besides the standard SPI lines: **IRQ** (referred as **INF_INT** in pin mapping section) line and **REQ** (**INF_I2C/_SPI** line reused after module startup). **IRQ** line is used for the module to indicate the host that there is data available to read or that the module is ready to start SPI transaction when host initiates the transaction by pulling **REQ** line down to indicate it wants to send/receive data. The purpose of using the **REQ** line with an interrupt reply is to act as a flow control, which allows the module to optimize power usage in ways that might compromise fast response to changes on the slave/chip select line. See the Host Initiated SPI transfer section for more details.

Otherwise, the SPI lines are mapped to the interface in the following way:

- **INF_SPI_CS**: SPI slave/chip select
- **INF_I2C_CLK/SPI_CLK**: SPI clock
- **INF_I2C_DAT/SPI_RX**: SPI MOSI
- **INF_SPI_TX**: SPI MISO

As in standard SPI communication, the host should keep the slave/chip select line high when it's not transmitting and keep it low during transmission. The other 3 lines behave in the standard manner as well: host clocks the clock line and uses MOSI to transmit to module and MISO for reading data from the module.

The length of transmission should be 32 bytes or the behavior is undefined.

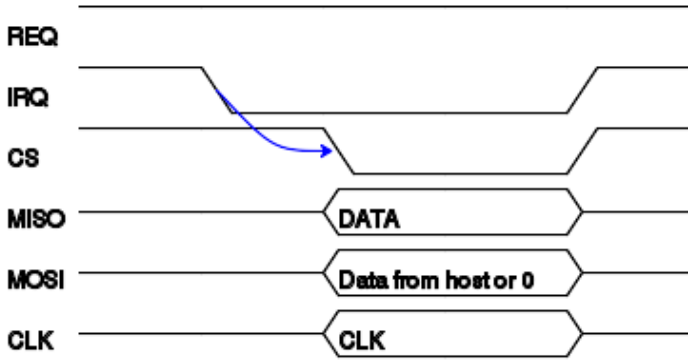
Note: There is no control for not reading data over SPI, which means that the host should check the received transmission for valid data whenever it has sent something to the module.

As with SPI both sides are transmitting, the host needs to make sure that the data transmitted by the host is zero in the case the host doesn't want to send commands to the module.

4.1.1 Module initiated SPI transfer

When the module indicates data availability by pulling the **IRQ** line down, a standard SPI transaction can be used immediately to retrieve the data from the module. The only constraint is the length of the transmission that should equal the constant length defined in section above.

The image below illustrates timings related to the SPI transaction in this case:



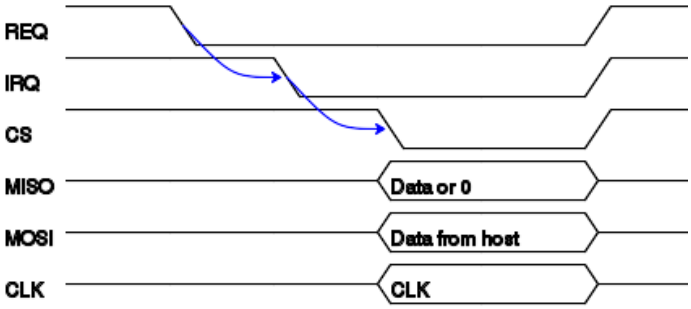
Module initiated SPI Transfer

4.1.2 Host Initiated SPI transfer

The REQ line exists to allow the host to initiate SPI transactions. When the line is pulled down, the module initializes the data to be transmitted to the host and pulls the IRQ line down. From this point onwards, the transaction behaves like in the first scenario. After the host has asserted the CS line, it should release the REQ line. If the host would like to transfer another frame, it should assert the REQ Line again after the CS is released.

Note: in this case, the module might transmit only zeros (0x00), indicating that there is no new data available from the module.

The image below illustrates timings for a SPI transaction in this case:



Host initiated SPI Transfer

5 Application example code structure

5.1 API

Messages are sent from the host to the module with the function send_to_module(hostMsg_t msg), and messages received by the host are passed to the function processFrame(hostMsg_t msg). The message format hostMsg_t is described in the section Protocol Data in the PulseOn Miniguide. In summary, the first byte is a message ID that determines what kind of data the message contains (see hostproto.h for the definitions of the IDs). A message whose name starts with GET causes a reply from the module whereas a message whose name starts with SET is handled quietly inside the module except when a SET message cannot be processed by the module. In that case the module replies with an COMMAND_STATUS message whose STATUS field is the ID of the type of error. For example, a message whose ID is not recognized causes the module to reply with a COMMAND_STATUS message whose STATUS field contains the error code ERROR_UNKNOWN_MSG_ID.

Four types of messages are sent by the module to the host without being requested.

- INST_ACTIVITY_DATA, ACCU_ACTIVITY_DATA, and SWIM_ACTIVITY_DATA, results based on accelerometer data, sent every ACTIVITY_INTERVAL milliseconds (default 1000)
- HR_DATA, results based on heart rate data, sent every HR_INTERVAL milliseconds (default 3000)
- IBI_DATA, the intervals between adjacent heart beats in milliseconds, sent every time a heart beat is detected

The INST_ACTIVITY_DATA, ACCU_ACTIVITY_DATA, and SWIM_ACTIVITY_DATA messages are sent in all modes, the HR_DATA message is sent in all modes except MODE_ON_DEMAND where it is sent only for a short time after the module receives REQUEST_HR, and the IBI_DATA message is sent in the modes MODE_SAMPLED and MODE_SLEEP. Note that the SWIM_ACTIVITY_DATA message is sent only after the module has received ENABLE_SWIM_ANALYSIS with the first byte set to a non-zero value. For more information about usage modes please refer to the section Applications and Usage modes in the PulseOn OHRM Miniguide. Only a few messages cannot be classified as setters, getters, or data messages. Among them are ARTIFACT_WARNING and REQUEST_HR.

The module is in idle mode after boot. This means that there is no activity in the module and no messages are sent to the host. The following code shows how to start the module in sports mode by sending a SET_USAGE_MODE to MODE_SPORTS.

```
hostMsg_t msg;
bzero(msg, sizeof(hostMsg_t));
setMsgId(msg, SET_USAGE_MODE);
setUsageMode(msg, MODE_SPORTS);

send_to_module(msg);
```

The PulseOn OHRM API including message ID's and their content are in the fllowing **hostproto.h** file. Included are also helper functions for parsing and building the content of a message.

5.2 hostTask

At beginning, hostTask resets the module by toggling GPIO pin connected to NRESET as 1 -> 0 - >1 . After reseting module, hostTask queues command to set module to Sports mode by setParam(SET_USAGE_MODE,MODE_SPORTS);

The host task implements receiving and sending of data in the hostTask main loop void hostTask(void *pvParameters) . After the initialization hostTask waits for module interrupt semaphore, if(xSemaphoreTake(irq_sem, 1000)).

Module informs host about data available by IRQ line 1->0 transition. Host Task implements interrupt handler for External Interrupt line GPIOTE_IRQHandler() in nrf51, EXTI1_IRQHandler(), in stm32. Processing done in Interrupt handler should be kept as short as possible. Interrupt handler clears GPIO port interrupt flag and signals irq semaphore xSemaphoreGiveFromISR(irq_sem, &xHigherPriorityTaskWoken).

When irq_sem is signaled, hostTask wakes up from semaphore wait (xSemaphoreTake), reserves SPI with spi_take() . Spi_take uses semaphore to prevent possible other tasks accessing SPI device same time than hostTask is using it. When SPI is reserved, hostTask issues CS to OHRM module, releases possible REQ line and performs SPI transfer.

If there are outgoing packets from host to the module, the hostTask sends them from the queue to the module with the same transfer it uses reading data from the module. If there are no outgoing messages in the queue, the host task just sends zero data. After the data is transferred, the hostTask releases the **CS** line and then releases SPI device.

Received data first byte not equal to zero indicates that hostTask received a valid frame and the frame is passed to messageHandler. In normal operation, we may receive a zero-frame from module when there is no data available but the host has initiated a transfer by the REQ line.

In raspberryPi the Interrupt handling is implemented in a different way. WiringPi interrupts are already user space interrupts so there is no need for using separate task or semaphore to activate user mode processing. In RaspberryPi user mode Interrupt handler calls readModule function that performs the same tasks as hostTask loop in FreeRTOS and then returns to the caller.

```
1
2
3
```

5.3 messageHandler

HostTask passes the frame received from the module to the messageHandler if the first byte of the frame is not zero. If the first byte is zero the frame does not contain any information, and it is not passed to the messageHandler. When the messageHandler receives the frame, it reads the first byte with getMsgId(msg) and processes it based on the message type as defined in hostproto.h. After the message is processed the messageHandler advances the pointer by the length of the processed message. A frame can contain more than one message, and subsequent messages in the frame are processed until the pointer is at the end of the frame, or the next message ID is zero. It is the responsibility of the host application's messageHandler to process all message IDs defined in hostproto.h, and to always advance the pointer along the frame in order to ensure messages are not missed.

In the example listed below the messages received from the module are printed to the serial console.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  #include "hostproto.h"
5  #include "hostMain.h"
6
7
8  void printDeviceStatus(hostMsg_t msg)
9  {
10   printf("messagehandler.c DEVICE_STATUS 0x%x\n", getDeviceStatusMode(msg));
11 }
12
13 void printCommandStatus(hostMsg_t msg)
14 {
15   printf("messagehandler.c COMMAND_STATUS command 0x%x, status 0x%x\n", getCommandS
16 }
17
18 void printUsageMode(hostMsg_t msg)
19 {
20   printf("messagehandler.c USAGE_MODE 0x%x\n", getUsageMode(msg));
21 }
22
23 void printSampledModeRatio(hostMsg_t msg)
24 {
25   printf("messagehandler.c SAMPLED_MODE_RATIO %d\n", getSampledModeRatio(msg));
26 }
27
28 void printHrInterval(hostMsg_t msg)
29 {
30   printf("messagehandler.c HR_INTERVAL %d\n", getHRInterval(msg));
```

```
}

void printActivityInterval(hostMsg_t msg)
{
  printf("messagehandler.c ACTIVITY_INTERVAL %d\n", getActivityInterval(msg));
}

void printUserSettings(hostMsg_t msg)
{
  printf("messagehandler.c USER_SETTINGS: msgId=%d timeStamp=%d ActivityLevel=%d "
        "Age=%d Height=%d Weight=%d Gender=%d MinHR=%d MaxHR=%d VO2max=%d\n",
        getMsgId(msg),
        getTstamp(msg),
        getUserActivityLevel(msg),
        getUserAge(msg),
        getUserHeight(msg),
        getUserWeight(msg),
        getUserGender(msg),
        getUserMinHR(msg),
        getUserMaxHR(msg),
        getUserVO2max(msg)
        /*,getFirstIteration(msg)*/);
}

void printInstActivityData(hostMsg_t msg)
{
  printf("messagehandler.c INST_ACTIVITY_DATA: msgId=0x%x timeStamp=%d WornState=%d
        getMsgId(msg),
        getTstamp(msg),
        getWornState(msg),
        getActivityClass(msg),
        getSleepClass(msg),
        getWorkoutClass(msg),
        getSpeedInertia(msg));
}

void printAccuActivityData(hostMsg_t msg)
{
  printf("messagehandler.c ACCU_ACTIVITY_DATA: msgId=0x%x timeStamp=%d WalkSteps=%d
        getMsgId(msg),
        getTstamp(msg),
        getWalkSteps(msg),
        getRunSteps(msg),
        getWalkDistance(msg),
        getRunDistance(msg ),
        getKCalInertia(msg));
}

void printHrData(hostMsg_t msg)
{
  printf("messagehandler.c HR_DATA: msgId=0x%x timeStamp=%d HR=%d HRQI=%d "
        "HROperatingStatus=%d MinHR=%d MaxHR=%d AvgHR=%d TrainingEffect=%d "
        "RelativeOC=%d KCal=%d VO2max=%d\n",
        getMsgId(msg),
        getTstamp(msg),
        getHR(msg),
```

```
        getHRQI(msg),
        getHROperatingStatus(msg),
        getMinHR(msg),
        getMaxHR(msg),
        getAvgHR(msg),
        getTrainingEffect(msg),
        getRelativeOC(msg),
        getKCal(msg),
        getVO2max(msg));
}

void printIbiData(hostMsg_t msg)
{
  printf("messagehandler.c IBI_DATA: msgId=0x%x timestamp=%d IBI=%d IBIQI=%d\n",
        getMsgId(msg),
        getTstamp(msg),
        getIBI(msg),
        getIBIQI(msg));
}


/**
 * @brief process message received from OHRM
 *
 *
 */

void processFrame(hostMsg_t msg)
{
  unsigned int i = 0; /* Current byte in the frame */
  int loopactive = 1;

  /* Loop the frame for messages */
  while (i < sizeof(hostMsg_t) && loopactive) {
    switch (getMsgId(msg + i)) {
      case DO_NOTHING: return; break;
      case INST_ACTIVITY_DATA: printInstActivityData(msg + i); break;
      case ACCU_ACTIVITY_DATA: printAccuActivityData(msg + i); break;
      case HR_DATA: printHrData(msg + i); break;
      case IBI_DATA: printIbiData(msg + i); break;
      case USER_SETTINGS: printUserSettings(msg + i); break;
      case USAGE_MODE: printUsageMode(msg + i); break;
      case SAMPLED_MODE_RATIO: printSampledModeRatio(msg + i); break;
      case HR_INTERVAL: printHrInterval(msg + i); break;
      case ACTIVITY_INTERVAL: printActivityInterval(msg + i); break;
      case DEVICE_STATUS: printDeviceStatus(msg + i); break;
      case COMMAND_STATUS: printCommandStatus(msg + i); break;

      /* Unknown message */
      default:
        loopactive = 0;
        printf("messagehandler.c unknown message ID %x\n", getMsgId(msg + i));
        break;
    }

    /* Increment i by message size */
```

```
143     i += getMsgSize(getMsgId(msg + i)) + 1;
144   }
145 }
```

5.4 hostMain

5.4.1 Building hostMain

HostMain needs enabled spi driver and wiringPi library to be installed.

5.4.1.1 Enable SPI

For more details on SPI on the Raspberry Pi look here

```
sudo raspi-config
```

select 8 advanced options A6 SPI and then enable SPI

Reboot the Raspberry Pi

5.4.1.2 Building wiringPi

```
cd wiringPi/
./build
```

5.4.1.3 Building hostMain

It can be built by invoking make in the source directory:

```
make
```

5.4.2 Using hostMain

The program hostMain.c for Raspberry allows quick testing of the communication protocol by using command line arguments to pass messages to the module. The program prints to the serial console the messages it sends and receives. To see the command line options, launch the executable (host) with the argument "-?".

Usage: host [OPTION]

Options:
-i Print hex dump of incoming messages (default: off)
-d Print hex dump of outgoing messages (default: off)

-m param_id:param_val
-r Reset module
-b Set boot mode (includes reset)
-? display this help and exit

The option -m specifies a command as listed in hostproto.h (note argument is in decimal format). For example "host -m64:1" puts the module in Sports Mode.

5.5 Example protocol dialog with module and nRF51 host

At line 4 host sends command 40 01 , set sports mode to device. At line 5 Device replies with Device status message 09. Lines 12, 14 are activity data messages from device (ID 01). Line 16 is heart rate message ID 02.

```
1  hostTask Started module reset
2  hostTask Started module reset done
3  send
4  40 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5  09 00 00 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
6  messagehandler.c DEVICE_STATUS
7  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8  09 00 00 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9  messagehandler.c DEVICE_STATUS
10 09 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11 messagehandler.c DEVICE_STATUS
12 01 04 80 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13 test_protocol.c ACTIVITY_DATA: msgId=1 timeStamp=1152 WornState=1 ActivityClass=1 WalkSte
14 01 08 89 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
15 test_protocol.c ACTIVITY_DATA: msgId=1 timeStamp=2185 WornState=1 ActivityClass=1 WalkSte
16 02 00 00 3d 53 00 3c 3d 3c 0a 14 00 98 00 28 00 00 00 00 00 00 00
17 test_protocol.c HR_DATA: msgId=2 timeStamp=0 HR=61 HRQI=83 HROperatingStatus=0 MinHR=
18 01 0c 71 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

5.6 nRF51 example app

nRF example app follows same architecture than block diagram in at beginning of document . Code is build top of nRF51-SDK_8.1.0, Softdevice S110 8.0, nRF51 SDK drivers and FreeRTOS. FreeRTOS is used abstraction of tast state handling and message passing. Other RTOS may be also used with minimal effort. This initial example is not yet using any softdevice BTLE services.

Code structure

- Makefile stantard Make for gcc in Mac/Linux
- main.c Set up HW and softdevice and start OHRM service task
- serail.c Interupt based UART driver that uses FreeRTOS queues
- spi.c SPI driver wrapper that uses FreeRTOS semaphores

- hostTask.c Host task described above
- messageHandler.c messageHandler described above
- error_handlers.c print error exceptions, this is not needed in production software
- printf-stdarg.c lean implementation of standard printf
- FreeRTOSConfig.h FreeRTOS configuratilns lke memory allocations etc
- hostproto.h PulseOn module host protocoll

nRF51 main.c initializes hardware including spi and uart and then starts OHRM tasks. doNothingTask() is just for demonstrating how you could add other tasks if needed.

```
1  /* NRF SDK Includes */
2  #include "nordic_common.h"
3  #include "softdevice_handler.h"
4  #include "nrf.h"
5  #include "nrf_gpio.h"
6
7  /* Scheduler include files. */
8  #include "FreeRTOS.h"
9  #include "task.h"
10 #include "queue.h"
11 #include "serial.h"
12 #include "hostTask.h"
13
14 #ifdef ENABLE_FLASH
15 #include "fat_sl.h"
16 #include "api_mdriver_flash.h"
17 #endif
18
19 #define LED_0      21
20 #define LED_1      22
21 #define LED_2      23
22 #define LED_3      24
23
24 /* Priorities at which the tasks are created. */
25 #define mainTASK_PRIORITY          ( tskIDLE_PRIORITY + 1 )
26 #define main_TASK_STACK_SIZE     ( configMINIMAL_STACK_SIZE + 160 )
27
28 void rprintf(const char *format,...);
29 void printf(const char *format,...);
30 void spi_init();
31
32 /**
33  * @brief Function called by APP_ERROR_HANDLER via APP_ERROR_CHECK macro if err_co
34  */
35 void app_error_handler(uint32_t error_code, uint32_t line_num, const uint8_t * p_file_name) {
36   for(;;);
37 }
38
39 /**
40  * @brief  Function called if an assert fails in the softdevice
41  */
42 void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
43 {
44   app_error_handler(0xDEADBEEF, line_num, p_file_name);
45 }
```

```
46
47  static void prvSetupHardware( void );
48
49  // Tasks
50  void hostTask( void *pvParameters );
51  void doNothingTask( void *pvParameters );
52
53  // Error status
54  unsigned portBASE_TYPE uxErrorStatus = pdPASS;
55
56  /**
57   * @brief main program, initialize system and start tasks
58   *
59   */
60  int main( void )
61  {
62    /* Setup the ports used by the demo and the clock. */
63    prvSetupHardware();
64
65    // Initialize serail port for console debug messages
66    xSerialPortInitMinimal(115200, 400);
67
68    // Initialize SPI used for communicationg with module
69    spi_init();
70
71    // Initialize host task parts that may referenced by other tasks
72    host_task_preinit();
73
74    // Start the tasks
75    xTaskCreate( doNothingTask, "doNothingTask", main_TASK_STACK_SIZE+100, NULL,mainTA
76    xTaskCreate( hostTask, "hostTask", main_TASK_STACK_SIZE+200, NULL,mainTASK_PRIORI
77
78    vTaskStartScheduler();
79
80    for( ;; );
81    return 0;
82  }
83
84
85  #define F_FS_THREAD_AWARE 1
86  /**
87   * @brief example how to add others tasks to system.
88   */
89  void doNothingTask( void *pvParameters )
90  {
91    int ret, j,i=0;
92    printf("KeepLiveTask started\n\r");
93
94  #ifdef ENABLE_FLASH
95    unsigned char status;
96    printf("Initializing Flash drive\n\r");
97    status =  f_initvolume( flash_initfunc );
98
99    if(status != F_NO_ERROR)
100   {
101     printError(status);
```

```
    // erase_drive();
    printf("Formatting drive to FAT12...\t");
    ret = f_hardformat(F_FAT12_MEDIA);
    if(ret)
        printError(ret);
    else
        printf("Done.\r\n");
     status =  f_initvolume( flash_initfunc );
    printf("f_initvolume = %d\n\r", status);

  }
  status = fs_init();
  if(status)
    printError(status);

  printf("Files: \n\r");
  listFiles("/*.*");
/*
  F_SPACE space;
  fn_getfreespace(&space);
  printf("Volume\tSize\tused\tavailable\tUse%\n\n\r");
  printf("vol1\t%d\t%d\t%d\t%f\n\r", space.total, space.free, space.used, (space.used/space.total)*
  fn_chdir("dir1");
  fn_delete("file1");
  fn_chdir("..");
  fn_rmdir("dir1");
  listFiles("/*.*");

  printf("creating directories:\n\r");
  printf("\tNew directory: \"dir1\"\r\n");
  fn_mkdir("dir1"); */
  printf("\n\rcreate a new file \"file1\"\n\r");
//  fn_chdir("dir1");
  F_FILE *file = fn_open("file1", "w");
  fn_write("contect of file1", 1, 17, file);
  fn_close(file);

//  fn_chdir("..");

  listFiles("/*.*");
#endif


    while(1)
    {
      vTaskDelay(200);
//      nrf_gpio_pin_toggle(LED_0);
//      printf("KeepLiveTask %d\n\r",i++);
      //      setParam(SET_USAGE_MODE,MODE_SPORTS);

    }
}

void heart_rate_meas_handler(int heart_rate)
{
}
```

```
158
159
160 /**
161  * @brief initialize Softdevice and PIO gor leds
162  */
163 void prvSetupHardware( void )
164 {
165   SOFTDEVICE_HANDLER_INIT(NRF_CLOCK_LFCLKSRC_XTAL_20_PPM, false);
166
167   nrf_gpio_pin_set(LED_0);
168   nrf_gpio_cfg_output(LED_0);
169   nrf_gpio_pin_set(LED_1);
170   nrf_gpio_cfg_output(LED_1);
171   nrf_gpio_pin_set(LED_2);
172   nrf_gpio_cfg_output(LED_2);
173   nrf_gpio_pin_set(LED_3);
174   nrf_gpio_cfg_output(LED_3);
175 }
176
177 /**
178  * @brief FreeRTOS enters here when there is no task to run
179  */
180 void vApplicationIdleHook( void ) {
181 __ASM volatile ("wfi");
182 //  uint32_t err_code = sd_app_evt_wait();
183     /* This signals the softdevice handler that we want the CPU to
184        sleep until an event/interrupt occurs. During this time the
185        softdevice will do what it needs to do; in our case: send
186        adverts */
187 //    APP_ERROR_CHECK(err_code);
188 }
189
```