/\* Copyright (c) 2014 Nordic Semiconductor. All Rights Reserved.

 \*

 \* The information contained herein is property of Nordic Semiconductor ASA.

 \* Terms and conditions of usage are described in detail in NORDIC

 \* SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.

 \*

 \* Licensees are granted free, non-transferable use of the information. NO

 \* WARRANTY of ANY KIND is provided. This heading must NOT be removed from

 \* the file.

 \*

 \*/

/\*\* @file

 \*

 \* @defgroup ble\_sdk\_app\_template\_main main.c

 \* @{

 \* @ingroup ble\_sdk\_app\_template

 \* @brief Template project main file.

 \*

 \* This file contains a template for creating a new application. It has the code necessary to wakeup

 \* from button, advertise, get a connection restart advertising on disconnect and if no new

 \* connection created go back to system-off mode.

 \* It can easily be used as a starting point for creating a new application, the comments identified

 \* with 'YOUR\_JOB' indicates where and how you can customize.

 \*/

//#include "Our\_Service.h"

///////////////////////

//#define BLE\_DFU\_APP\_SUPPORT 1;

#include <stdint.h>

#include <string.h>

#include "nordic\_common.h"

#include "nrf.h"

#include "app\_error.h"

#include "nrf51\_bitfields.h"

#include "ble.h"

#include "ble\_hci.h"

#include "ble\_srv\_common.h"

#include "ble\_advdata.h"

#include "ble\_advertising.h"

#include "ble\_conn\_params.h"

#include "boards.h"

#include "softdevice\_handler.h"

#include "app\_timer.h"

#include "device\_manager.h"

#include "pstorage.h"

#include "app\_trace.h"

#include "bsp.h"

#include "bsp\_btn\_ble.h"

#include "SEGGER\_RTT.h"

#ifdef BLE\_DFU\_APP\_SUPPORT

#include "ble\_dfu.h"

#include "dfu\_app\_handler.h"

#include "nrf\_delay.h"

#endif // BLE\_DFU\_APP\_SUPPORT

//For the custom service

#define BLE\_UUID\_OUR\_BASE\_UUID\_VS {0x23, 0xD1, 0x13, 0xEF, 0x5F, 0x78, 0x23, 0x15, 0xDE, 0xEF, 0x12, 0x12, 0x00, 0x00, 0x00, 0x00} // 128-bit base UUID

#define BLE\_UUID\_OUR\_SERVICE\_UUID 0xABCD // Just a random, but recognizable value

typedef struct

{

 uint16\_t service\_handle; /\*\*< Handle of Our Service (as provided by the BLE stack). \*/

}ble\_os\_t;

///////////////////////

//#define BLE\_DFU\_APP\_SUPPORT 1;

#define IS\_SRVC\_CHANGED\_CHARACT\_PRESENT 1 /\*\*< Include or not the service\_changed characteristic. if not enabled, the server's database cannot be changed for the lifetime of the device\*/

#define DEVICE\_NAME "Nordic\_Template" /\*\*< Name of device. Will be included in the advertising data. \*/

#define MANUFACTURER\_NAME "NordicSemiconductor" /\*\*< Manufacturer. Will be passed to Device Information Service. \*/

#define APP\_ADV\_INTERVAL 300 /\*\*< The advertising interval (in units of 0.625 ms. This value corresponds to 25 ms). \*/

#define APP\_ADV\_TIMEOUT\_IN\_SECONDS 180 /\*\*< The advertising timeout in units of seconds. \*/

#define APP\_TIMER\_PRESCALER 0 /\*\*< Value of the RTC1 PRESCALER register. \*/

#define APP\_TIMER\_MAX\_TIMERS (6+BSP\_APP\_TIMERS\_NUMBER) /\*\*< Maximum number of simultaneously created timers. \*/

#define APP\_TIMER\_OP\_QUEUE\_SIZE 4 /\*\*< Size of timer operation queues. \*/

#define LED\_FLIP\_INTERVAL APP\_TIMER\_TICKS(1000, APP\_TIMER\_PRESCALER)

#define MIN\_FLIP\_RATE 140

#define MAX\_FLIP\_RATE 300

#define FLIP\_INCREMENT `0

//static sensorsim\_cfg\_t m\_battery\_sim\_cfg;

#define MIN\_CONN\_INTERVAL MSEC\_TO\_UNITS(100, UNIT\_1\_25\_MS) /\*\*< Minimum acceptable connection interval (0.1 seconds). \*/

#define MAX\_CONN\_INTERVAL MSEC\_TO\_UNITS(200, UNIT\_1\_25\_MS) /\*\*< Maximum acceptable connection interval (0.2 second). \*/

#define SLAVE\_LATENCY 0 /\*\*< Slave latency. \*/

#define CONN\_SUP\_TIMEOUT MSEC\_TO\_UNITS(4000, UNIT\_10\_MS) /\*\*< Connection supervisory timeout (4 seconds). \*/

#define FIRST\_CONN\_PARAMS\_UPDATE\_DELAY APP\_TIMER\_TICKS(5000, APP\_TIMER\_PRESCALER) /\*\*< Time from initiating event (connect or start of notification) to first time sd\_ble\_gap\_conn\_param\_update is called (5 seconds). \*/

#define NEXT\_CONN\_PARAMS\_UPDATE\_DELAY APP\_TIMER\_TICKS(30000, APP\_TIMER\_PRESCALER)/\*\*< Time between each call to sd\_ble\_gap\_conn\_param\_update after the first call (30 seconds). \*/

#define MAX\_CONN\_PARAMS\_UPDATE\_COUNT 3 /\*\*< Number of attempts before giving up the connection parameter negotiation. \*/

#define SEC\_PARAM\_BOND 1 /\*\*< Perform bonding. \*/

#define SEC\_PARAM\_MITM 0 /\*\*< Man In The Middle protection not required. \*/

#define SEC\_PARAM\_IO\_CAPABILITIES BLE\_GAP\_IO\_CAPS\_NONE /\*\*< No I/O capabilities. \*/

#define SEC\_PARAM\_OOB 0 /\*\*< Out Of Band data not available. \*/

#define SEC\_PARAM\_MIN\_KEY\_SIZE 7 /\*\*< Minimum encryption key size. \*/

#define SEC\_PARAM\_MAX\_KEY\_SIZE 16 /\*\*< Maximum encryption key size. \*/

#define DEAD\_BEEF 0xDEADBEEF /\*\*< Value used as error code on stack dump, can be used to identify stack location on stack unwind. \*/

#ifdef BLE\_DFU\_APP\_SUPPORT

#define DFU\_REV\_MAJOR 0x00 /\*\* DFU Major revision number to be exposed. \*/

#define DFU\_REV\_MINOR 0x01 /\*\* DFU Minor revision number to be exposed. \*/

#define DFU\_REVISION ((DFU\_REV\_MAJOR << 8) | DFU\_REV\_MINOR) /\*\* DFU Revision number to be exposed. Combined of major and minor versions. \*/

#define APP\_SERVICE\_HANDLE\_START 0x000C /\*\*< Handle of first application specific service when when service changed characteristic is present. \*/

#define BLE\_HANDLE\_MAX 0xFFFF /\*\*< Max handle value in BLE. \*/

STATIC\_ASSERT(IS\_SRVC\_CHANGED\_CHARACT\_PRESENT); /\*\* When having DFU Service support in application the Service Changed Characteristic should always be present. \*/

#endif // BLE\_DFU\_APP\_SUPPORT

static dm\_application\_instance\_t m\_app\_handle; /\*\*< Application identifier allocated by device manager \*/

static uint16\_t m\_conn\_handle = BLE\_CONN\_HANDLE\_INVALID; /\*\*< Handle of the current connection. \*/

/\* YOUR\_JOB: Declare all services structure your application is using

static ble\_xx\_service\_t m\_xxs;

static ble\_yy\_service\_t m\_yys;

\*/

static ble\_os\_t our\_service\_info; //actually it might be handling it

// YOUR\_JOB: Use UUIDs for service(s) used in your application.

static ble\_uuid\_t m\_adv\_uuids[] = {{BLE\_UUID\_DEVICE\_INFORMATION\_SERVICE, BLE\_UUID\_TYPE\_BLE}}; /\*\*< Universally unique service identifiers. \*/

#ifdef BLE\_DFU\_APP\_SUPPORT

static ble\_dfu\_t m\_dfus; /\*\*< Structure used to identify the DFU service. \*/

#endif // BLE\_DFU\_APP\_SUPPORT

/\*\*@brief Callback function for asserts in the SoftDevice.

 \*

 \* @details This function will be called in case of an assert in the SoftDevice.

 \*

 \* @warning This handler is an example only and does not fit a final product. You need to analyze

 \* how your product is supposed to react in case of Assert.

 \* @warning On assert from the SoftDevice, the system can only recover on reset.

 \*

 \* @param[in] line\_num Line number of the failing ASSERT call.

 \* @param[in] file\_name File name of the failing ASSERT call.

 \*/

void assert\_nrf\_callback(uint16\_t line\_num, const uint8\_t \* p\_file\_name)

{

 app\_error\_handler(DEAD\_BEEF, line\_num, p\_file\_name);

}

#ifdef BLE\_DFU\_APP\_SUPPORT

/\*\*@brief Function for stopping advertising.

 \*/

static void advertising\_stop(void)

{

 uint32\_t err\_code;

 err\_code = sd\_ble\_gap\_adv\_stop();

 APP\_ERROR\_CHECK(err\_code);

 err\_code = bsp\_indication\_set(BSP\_INDICATE\_IDLE);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for loading application-specific context after establishing a secure connection.

 \*

 \* @details This function will load the application context and check if the ATT table is marked as

 \* changed. If the ATT table is marked as changed, a Service Changed Indication

 \* is sent to the peer if the Service Changed CCCD is set to indicate.

 \*

 \* @param[in] p\_handle The Device Manager handle that identifies the connection for which the context

 \* should be loaded.

 \*/

static void app\_context\_load(dm\_handle\_t const \* p\_handle)

{

 uint32\_t err\_code;

 static uint32\_t context\_data;

 dm\_application\_context\_t context;

 context.len = sizeof(context\_data);

 context.p\_data = (uint8\_t \*)&context\_data;

 err\_code = dm\_application\_context\_get(p\_handle, &context);

 if (err\_code == NRF\_SUCCESS)

 {

 // Send Service Changed Indication if ATT table has changed.

 if ((context\_data & (DFU\_APP\_ATT\_TABLE\_CHANGED << DFU\_APP\_ATT\_TABLE\_POS)) != 0)

 {

 err\_code = sd\_ble\_gatts\_service\_changed(m\_conn\_handle, APP\_SERVICE\_HANDLE\_START, BLE\_HANDLE\_MAX);

 if ((err\_code != NRF\_SUCCESS) &&

 (err\_code != BLE\_ERROR\_INVALID\_CONN\_HANDLE) &&

 (err\_code != NRF\_ERROR\_INVALID\_STATE) &&

 (err\_code != BLE\_ERROR\_NO\_TX\_BUFFERS) &&

 (err\_code != NRF\_ERROR\_BUSY) &&

 (err\_code != BLE\_ERROR\_GATTS\_SYS\_ATTR\_MISSING))

 {

 APP\_ERROR\_HANDLER(err\_code);

 }

 }

 err\_code = dm\_application\_context\_delete(p\_handle);

 APP\_ERROR\_CHECK(err\_code);

 }

 else if (err\_code == DM\_NO\_APP\_CONTEXT)

 {

 // No context available. Ignore.

 }

 else

 {

 APP\_ERROR\_HANDLER(err\_code);

 }

}

/\*\* @snippet [DFU BLE Reset prepare] \*/

/\*\*@brief Function for preparing for system reset.

 \*

 \* @details This function implements @ref dfu\_app\_reset\_prepare\_t. It will be called by

 \* @ref dfu\_app\_handler.c before entering the bootloader/DFU.

 \* This allows the current running application to shut down gracefully.

 \*/

static void reset\_prepare(void)

{

 uint32\_t err\_code;

 if (m\_conn\_handle != BLE\_CONN\_HANDLE\_INVALID)

 {

 // Disconnect from peer.

 err\_code = sd\_ble\_gap\_disconnect(m\_conn\_handle, BLE\_HCI\_REMOTE\_USER\_TERMINATED\_CONNECTION);

 APP\_ERROR\_CHECK(err\_code);

 err\_code = bsp\_indication\_set(BSP\_INDICATE\_IDLE);

 APP\_ERROR\_CHECK(err\_code);

 }

 else

 {

 // If not connected, the device will be advertising. Hence stop the advertising.

 advertising\_stop();

 }

 err\_code = ble\_conn\_params\_stop();

 APP\_ERROR\_CHECK(err\_code);

 nrf\_delay\_ms(500);

}

/\*\* @snippet [DFU BLE Reset prepare] \*/

#endif // BLE\_DFU\_APP\_SUPPORT

/\*\*@brief Function for the Timer initialization.

 \*

 \* @details Initializes the timer module. This creates and starts application timers.

 \*/

static void timers\_init(void)

{

 // Initialize timer module.

 APP\_TIMER\_INIT(APP\_TIMER\_PRESCALER, APP\_TIMER\_MAX\_TIMERS, APP\_TIMER\_OP\_QUEUE\_SIZE, false);

 // Create timers.

 /\* YOUR\_JOB: Create any timers to be used by the application.

 Below is an example of how to create a timer.

 For every new timer needed, increase the value of the macro APP\_TIMER\_MAX\_TIMERS by

 one.

 uint32\_t err\_code;

 err\_code = app\_timer\_create(&m\_app\_timer\_id, APP\_TIMER\_MODE\_REPEATED, timer\_timeout\_handler);

 APP\_ERROR\_CHECK(err\_code); \*/

}

/\*\*@brief Function for the GAP initialization.

 \*

 \* @details This function sets up all the necessary GAP (Generic Access Profile) parameters of the

 \* device including the device name, appearance, and the preferred connection parameters.

 \*/

static void gap\_params\_init(void)

{

 uint32\_t err\_code;

 ble\_gap\_conn\_params\_t gap\_conn\_params;

 ble\_gap\_conn\_sec\_mode\_t sec\_mode;

 BLE\_GAP\_CONN\_SEC\_MODE\_SET\_OPEN(&sec\_mode);

 err\_code = sd\_ble\_gap\_device\_name\_set(&sec\_mode,

 (const uint8\_t \*)DEVICE\_NAME,

 strlen(DEVICE\_NAME));

 APP\_ERROR\_CHECK(err\_code);

 //err\_code = sd\_ble\_gap\_appearance\_set(0x0140);

 //APP\_ERROR\_CHECK(err\_code);

 /\* YOUR\_JOB: Use an appearance value matching the application's use case.

 err\_code = sd\_ble\_gap\_appearance\_set(BLE\_APPEARANCE\_);

 APP\_ERROR\_CHECK(err\_code); \*/

 memset(&gap\_conn\_params, 0, sizeof(gap\_conn\_params));

 gap\_conn\_params.min\_conn\_interval = MIN\_CONN\_INTERVAL;

 gap\_conn\_params.max\_conn\_interval = MAX\_CONN\_INTERVAL;

 gap\_conn\_params.slave\_latency = SLAVE\_LATENCY;

 gap\_conn\_params.conn\_sup\_timeout = CONN\_SUP\_TIMEOUT;

 err\_code = sd\_ble\_gap\_ppcp\_set(&gap\_conn\_params);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for handling the YYY Service events.

 \* YOUR\_JOB implement a service handler function depending on the event the service you are using can generate

 \*

 \* @details This function will be called for all YY Service events which are passed to

 \* the application.

 \*

 \* @param[in] p\_yy\_service YY Service structure.

 \* @param[in] p\_evt Event received from the YY Service.

 \*

 \*

static void on\_yys\_evt(ble\_yy\_service\_t \* p\_yy\_service,

 ble\_yy\_service\_evt\_t \* p\_evt)

{

 switch (p\_evt->evt\_type)

 {

 case BLE\_YY\_NAME\_EVT\_WRITE:

 APPL\_LOG("[APPL]: charact written with value %s. \r\n", p\_evt->params.char\_xx.value.p\_str);

 break;

 default:

 // No implementation needed.

 break;

 }

}\*/

void our\_service\_init(ble\_os\_t \*our\_service\_info){

 //implementing the method

 uint32\_t err\_code;

 ble\_uuid\_t service\_uuid;

 service\_uuid.uuid = BLE\_UUID\_OUR\_SERVICE\_UUID;

 ble\_uuid128\_t base\_uuid = BLE\_UUID\_OUR\_BASE\_UUID\_VS;

 //register the Vender Specific (V.S.) UUID with the BLE Stack

 //now we can use the 16 bit UUID, I think

 err\_code = sd\_ble\_uuid\_vs\_add(&base\_uuid, &service\_uuid.type);

 APP\_ERROR\_CHECK(err\_code); //error check.

 //service is now registered with the BLE stack

 //initialize the service

 err\_code = sd\_ble\_gatts\_service\_add(BLE\_GATTS\_SRVC\_TYPE\_PRIMARY, &service\_uuid,

 &our\_service\_info->service\_handle);

 //the above might be attaching the the handle to that particlar service

 APP\_ERROR\_CHECK(err\_code);

 //Open up the RTT terminal for debugging. There is an article on it on Tutorials website. Real Time Debugging, I think

 //this is not necessary though

 SEGGER\_RTT\_WriteString(0, "Our specific service was relatively recently initialized as a primary service");

unsigned char user\_description[] = "Yay";

ble\_gatts\_char\_md\_t our\_characteristic;

our\_characteristic.char\_props.write\_wo\_resp = 1;

our\_characteristic.char\_user\_desc\_max\_size = 4;

our\_characteristic.p\_char\_user\_desc = user\_description;

our\_characteristic.p\_char\_pf= NULL;

our\_characteristic.char\_user\_desc\_size=4;

our\_characteristic.p\_user\_desc\_md = NULL;

our\_characteristic.p\_cccd\_md = NULL;

our\_characteristic.p\_sccd\_md = NULL;

ble\_gatts\_attr\_t our\_attribute;

memset(&our\_attribute, 0, sizeof(our\_attribute));

//this is one thing I am thinking of adding

//our\_attribute.p\_uuid = ; // ble\_srv\_common.c in the ble\_app\_hrs application s130 with DFU information

// our\_attribute.p\_attr\_md = &attr\_md;

// our\_attribute.max\_len = p\_char\_props->max\_len;

// our\_attribute.

ble\_gatts\_char\_handles\_t our\_characteristic\_handle; //This is another thing I am thinking of adding.

memset(&our\_characteristic\_handle, 0, sizeof(our\_characteristic\_handle));

////our\_characteristic

////Characteristic added

err\_code = sd\_ble\_gatts\_characteristic\_add(our\_service\_info->service\_handle, &our\_characteristic,&our\_attribute, &our\_characteristic\_handle);

//APP\_ERROR\_CHECK(err\_code);

//DEBUGGING : It appears that the characteristic is not being added to the service we did.

if(err\_code == NRF\_ERROR\_INVALID\_ADDR){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_INVALID\_ADDR\n"); //This one is it. //This makes sense because I have not done research on adding a characterisitc.

}

else if(err\_code ==NRF\_ERROR\_INVALID\_PARAM){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_INVALID\_PARAM\n");

}

else if(err\_code ==NRF\_ERROR\_INVALID\_STATE){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_INVALID\_STATE!\n");

}

else if(err\_code ==NRF\_ERROR\_FORBIDDEN){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_FORBIDDEN\n");

}

else if(err\_code ==NRF\_ERROR\_NO\_MEM){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_NO\_MEM\n");

}

else if(err\_code ==NRF\_ERROR\_DATA\_SIZE){

 SEGGER\_RTT\_WriteString(0, "NRF\_ERROR\_DATA\_SIZE\n");

}

}

/\*\*@brief Function for initializing services that will be used by the application.

 \*/

static void services\_init(void)

{

 our\_service\_init(&our\_service\_info);

 uint32\_t err\_code;

 /\* YOUR\_JOB: Add code to initialize the services used by the application.

 uint32\_t err\_code;

 ble\_xxs\_init\_t xxs\_init;

 ble\_yys\_init\_t yys\_init;

 // Initialize XXX Service.

 memset(&xxs\_init, 0, sizeof(xxs\_init));

 xxs\_init.evt\_handler = NULL;

 xxs\_init.is\_xxx\_notify\_supported = true;

 xxs\_init.ble\_xx\_initial\_value.level = 100;

 err\_code = ble\_bas\_init(&m\_xxs, &xxs\_init);

 APP\_ERROR\_CHECK(err\_code);

 // Initialize YYY Service.

 memset(&yys\_init, 0, sizeof(yys\_init));

 yys\_init.evt\_handler = on\_yys\_evt;

 yys\_init.ble\_yy\_initial\_value.counter = 0;

 err\_code = ble\_yy\_service\_init(&yys\_init, &yy\_init);

 APP\_ERROR\_CHECK(err\_code);

 \*/

 #ifdef BLE\_DFU\_APP\_SUPPORT

 /\*\* @snippet [DFU BLE Service initialization] \*/

 ble\_dfu\_init\_t dfus\_init;

 //taken from online

 SEGGER\_RTT\_WriteString(0, "DFU SERVICE INITIALIZED!!!\n");

 // Initialize the Device Firmware Update Service.

 memset(&dfus\_init, 0, sizeof(dfus\_init));

 dfus\_init.evt\_handler = dfu\_app\_on\_dfu\_evt;

 dfus\_init.error\_handler = NULL;

 dfus\_init.evt\_handler = dfu\_app\_on\_dfu\_evt;

 dfus\_init.revision = DFU\_REVISION;

 err\_code = ble\_dfu\_init(&m\_dfus, &dfus\_init);

 APP\_ERROR\_CHECK(err\_code);

 dfu\_app\_reset\_prepare\_set(reset\_prepare);

 dfu\_app\_dm\_appl\_instance\_set(m\_app\_handle);

 /\*\* @snippet [DFU BLE Service initialization] \*/

#endif // BLE\_DFU\_APP\_SUPPORT

}

/\*\*@brief Function for handling the Connection Parameters Module.

 \*

 \* @details This function will be called for all events in the Connection Parameters Module which

 \* are passed to the application.

 \* @note All this function does is to disconnect. This could have been done by simply

 \* setting the disconnect\_on\_fail config parameter, but instead we use the event

 \* handler mechanism to demonstrate its use.

 \*

 \* @param[in] p\_evt Event received from the Connection Parameters Module.

 \*/

static void on\_conn\_params\_evt(ble\_conn\_params\_evt\_t \* p\_evt)

{

 uint32\_t err\_code;

 if (p\_evt->evt\_type == BLE\_CONN\_PARAMS\_EVT\_FAILED)

 {

 err\_code = sd\_ble\_gap\_disconnect(m\_conn\_handle, BLE\_HCI\_CONN\_INTERVAL\_UNACCEPTABLE);

 APP\_ERROR\_CHECK(err\_code);

 }

}

/\*\*@brief Function for handling a Connection Parameters error.

 \*

 \* @param[in] nrf\_error Error code containing information about what went wrong.

 \*/

static void conn\_params\_error\_handler(uint32\_t nrf\_error)

{

 APP\_ERROR\_HANDLER(nrf\_error);

}

/\*\*@brief Function for initializing the Connection Parameters module.

 \*/

static void conn\_params\_init(void)

{

 uint32\_t err\_code;

 ble\_conn\_params\_init\_t cp\_init;

 memset(&cp\_init, 0, sizeof(cp\_init));

 cp\_init.p\_conn\_params = NULL;

 cp\_init.first\_conn\_params\_update\_delay = FIRST\_CONN\_PARAMS\_UPDATE\_DELAY;

 cp\_init.next\_conn\_params\_update\_delay = NEXT\_CONN\_PARAMS\_UPDATE\_DELAY;

 cp\_init.max\_conn\_params\_update\_count = MAX\_CONN\_PARAMS\_UPDATE\_COUNT;

 cp\_init.start\_on\_notify\_cccd\_handle = BLE\_GATT\_HANDLE\_INVALID;

 cp\_init.disconnect\_on\_fail = false;

 cp\_init.evt\_handler = on\_conn\_params\_evt;

 cp\_init.error\_handler = conn\_params\_error\_handler;

 err\_code = ble\_conn\_params\_init(&cp\_init);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for starting timers.

\*/

static void application\_timers\_start(void)

{

 /\* YOUR\_JOB: Start your timers. below is an example of how to start a timer.

 uint32\_t err\_code;

 err\_code = app\_timer\_start(m\_app\_timer\_id, TIMER\_INTERVAL, NULL);

 APP\_ERROR\_CHECK(err\_code); \*/

}

/\*\*@brief Function for putting the chip into sleep mode.

 \*

 \* @note This function will not return.

 \*/

static void sleep\_mode\_enter(void)

{

 uint32\_t err\_code = bsp\_indication\_set(BSP\_INDICATE\_IDLE);

 APP\_ERROR\_CHECK(err\_code);

 // Prepare wakeup buttons.

 err\_code = bsp\_btn\_ble\_sleep\_mode\_prepare();

 APP\_ERROR\_CHECK(err\_code);

 // Go to system-off mode (this function will not return; wakeup will cause a reset).

 err\_code = sd\_power\_system\_off();

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for handling advertising events.

 \*

 \* @details This function will be called for advertising events which are passed to the application.

 \*

 \* @param[in] ble\_adv\_evt Advertising event.

 \*/

static void on\_adv\_evt(ble\_adv\_evt\_t ble\_adv\_evt)

{

 uint32\_t err\_code;

 switch (ble\_adv\_evt)

 {

 case BLE\_ADV\_EVT\_FAST:

 err\_code = bsp\_indication\_set(BSP\_INDICATE\_ADVERTISING);

 APP\_ERROR\_CHECK(err\_code);

 break;

 case BLE\_ADV\_EVT\_IDLE:

 sleep\_mode\_enter();

 break;

 default:

 break;

 }

}

/\*\*@brief Function for handling the Application's BLE Stack events.

 \*

 \* @param[in] p\_ble\_evt Bluetooth stack event.

 \*/

static void on\_ble\_evt(ble\_evt\_t \* p\_ble\_evt)

{

 uint32\_t err\_code;

 switch (p\_ble\_evt->header.evt\_id)

 {

 case BLE\_GAP\_EVT\_CONNECTED:

 err\_code = bsp\_indication\_set(BSP\_INDICATE\_CONNECTED);

 APP\_ERROR\_CHECK(err\_code);

 m\_conn\_handle = p\_ble\_evt->evt.gap\_evt.conn\_handle;

 break;

 case BLE\_GAP\_EVT\_DISCONNECTED:

 m\_conn\_handle = BLE\_CONN\_HANDLE\_INVALID;

 break;

 default:

 // No implementation needed.

 break;

 }

}

/\*\*@brief Function for dispatching a BLE stack event to all modules with a BLE stack event handler.

 \*

 \* @details This function is called from the BLE Stack event interrupt handler after a BLE stack

 \* event has been received.

 \*

 \* @param[in] p\_ble\_evt Bluetooth stack event.

 \*/

static void ble\_evt\_dispatch(ble\_evt\_t \* p\_ble\_evt)

{

 dm\_ble\_evt\_handler(p\_ble\_evt);

 ble\_conn\_params\_on\_ble\_evt(p\_ble\_evt);

 bsp\_btn\_ble\_on\_ble\_evt(p\_ble\_evt);

 #ifdef BLE\_DFU\_APP\_SUPPORT

 /\*\* @snippet [Propagating BLE Stack events to DFU Service] \*/

 ble\_dfu\_on\_ble\_evt(&m\_dfus, p\_ble\_evt);

 /\*\* @snippet [Propagating BLE Stack events to DFU Service] \*/

 #endif // BLE\_DFU\_APP\_SUPPORT

 on\_ble\_evt(p\_ble\_evt);

 ble\_advertising\_on\_ble\_evt(p\_ble\_evt);

 /\*YOUR\_JOB add calls to \_on\_ble\_evt functions from each service your application is using

 ble\_xxs\_on\_ble\_evt(&m\_xxs, p\_ble\_evt);

 ble\_yys\_on\_ble\_evt(&m\_yys, p\_ble\_evt);

 \*/

}

/\*\*@brief Function for dispatching a system event to interested modules.

 \*

 \* @details This function is called from the System event interrupt handler after a system

 \* event has been received.

 \*

 \* @param[in] sys\_evt System stack event.

 \*/

static void sys\_evt\_dispatch(uint32\_t sys\_evt)

{

 pstorage\_sys\_event\_handler(sys\_evt);

 ble\_advertising\_on\_sys\_evt(sys\_evt);

}

/\*\*@brief Function for initializing the BLE stack.

 \*

 \* @details Initializes the SoftDevice and the BLE event interrupt.

 \*/

static void ble\_stack\_init(void)

{

 uint32\_t err\_code;

 // Initialize the SoftDevice handler module.

 SOFTDEVICE\_HANDLER\_INIT(NRF\_CLOCK\_LFCLKSRC\_XTAL\_20\_PPM, NULL);

#if defined(S110) || defined(S130)

 // Enable BLE stack.

 ble\_enable\_params\_t ble\_enable\_params;

 memset(&ble\_enable\_params, 0, sizeof(ble\_enable\_params));

#ifdef S130

 ble\_enable\_params.gatts\_enable\_params.attr\_tab\_size = BLE\_GATTS\_ATTR\_TAB\_SIZE\_DEFAULT;

#endif

 ble\_enable\_params.gatts\_enable\_params.service\_changed = IS\_SRVC\_CHANGED\_CHARACT\_PRESENT;

 err\_code = sd\_ble\_enable(&ble\_enable\_params);

 APP\_ERROR\_CHECK(err\_code);

#endif

 // Register with the SoftDevice handler module for BLE events.

 err\_code = softdevice\_ble\_evt\_handler\_set(ble\_evt\_dispatch);

 APP\_ERROR\_CHECK(err\_code);

 // Register with the SoftDevice handler module for BLE events.

 err\_code = softdevice\_sys\_evt\_handler\_set(sys\_evt\_dispatch);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for handling events from the BSP module.

 \*

 \* @param[in] event Event generated by button press.

 \*/

void bsp\_event\_handler(bsp\_event\_t event)

{

 uint32\_t err\_code;

 switch (event)

 {

 case BSP\_EVENT\_SLEEP:

 sleep\_mode\_enter();

 break;

 case BSP\_EVENT\_DISCONNECT:

 err\_code = sd\_ble\_gap\_disconnect(m\_conn\_handle, BLE\_HCI\_REMOTE\_USER\_TERMINATED\_CONNECTION);

 if (err\_code != NRF\_ERROR\_INVALID\_STATE)

 {

 APP\_ERROR\_CHECK(err\_code);

 }

 break;

 case BSP\_EVENT\_WHITELIST\_OFF:

 err\_code = ble\_advertising\_restart\_without\_whitelist();

 if (err\_code != NRF\_ERROR\_INVALID\_STATE)

 {

 APP\_ERROR\_CHECK(err\_code);

 }

 break;

 default:

 break;

 }

}

/\*\*@brief Function for handling the Device Manager events.

 \*

 \* @param[in] p\_evt Data associated to the device manager event.

 \*/

static uint32\_t device\_manager\_evt\_handler(dm\_handle\_t const \* p\_handle,

 dm\_event\_t const \* p\_event,

 ret\_code\_t event\_result)

{

 APP\_ERROR\_CHECK(event\_result);

#ifdef BLE\_DFU\_APP\_SUPPORT

 if (p\_event->event\_id == DM\_EVT\_LINK\_SECURED)

 {

 app\_context\_load(p\_handle);

 }

#endif // BLE\_DFU\_APP\_SUPPORT

 return NRF\_SUCCESS;

}

/\*\*@brief Function for the Device Manager initialization.

 \*

 \* @param[in] erase\_bonds Indicates whether bonding information should be cleared from

 \* persistent storage during initialization of the Device Manager.

 \*/

static void device\_manager\_init(bool erase\_bonds)

{

 uint32\_t err\_code;

 dm\_init\_param\_t init\_param = {.clear\_persistent\_data = erase\_bonds};

 dm\_application\_param\_t register\_param;

 // Initialize persistent storage module.

 err\_code = pstorage\_init();

 APP\_ERROR\_CHECK(err\_code);

 err\_code = dm\_init(&init\_param);

 APP\_ERROR\_CHECK(err\_code);

 memset(&register\_param.sec\_param, 0, sizeof(ble\_gap\_sec\_params\_t));

 register\_param.sec\_param.bond = SEC\_PARAM\_BOND;

 register\_param.sec\_param.mitm = SEC\_PARAM\_MITM;

 register\_param.sec\_param.io\_caps = SEC\_PARAM\_IO\_CAPABILITIES;

 register\_param.sec\_param.oob = SEC\_PARAM\_OOB;

 register\_param.sec\_param.min\_key\_size = SEC\_PARAM\_MIN\_KEY\_SIZE;

 register\_param.sec\_param.max\_key\_size = SEC\_PARAM\_MAX\_KEY\_SIZE;

 register\_param.evt\_handler = device\_manager\_evt\_handler;

 register\_param.service\_type = DM\_PROTOCOL\_CNTXT\_GATT\_SRVR\_ID;

 err\_code = dm\_register(&m\_app\_handle, &register\_param);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for initializing the Advertising functionality.

 \*/

static void advertising\_init(void)

{

 uint32\_t err\_code;

 ble\_advdata\_t advdata;

 // Build advertising data struct to pass into @ref ble\_advertising\_init.

 memset(&advdata, 0, sizeof(advdata));

 advdata.name\_type = BLE\_ADVDATA\_FULL\_NAME;

 advdata.include\_appearance = true;

 advdata.flags = BLE\_GAP\_ADV\_FLAGS\_LE\_ONLY\_GENERAL\_DISC\_MODE;

 advdata.uuids\_complete.uuid\_cnt = sizeof(m\_adv\_uuids) / sizeof(m\_adv\_uuids[0]);

 advdata.uuids\_complete.p\_uuids = m\_adv\_uuids;

 ble\_adv\_modes\_config\_t options = {0};

 options.ble\_adv\_fast\_enabled = BLE\_ADV\_FAST\_ENABLED;

 options.ble\_adv\_fast\_interval = APP\_ADV\_INTERVAL;

 options.ble\_adv\_fast\_timeout = APP\_ADV\_TIMEOUT\_IN\_SECONDS;

 err\_code = ble\_advertising\_init(&advdata, NULL, &options, on\_adv\_evt, NULL);

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for initializing buttons and leds.

 \*

 \* @param[out] p\_erase\_bonds Will be true if the clear bonding button was pressed to wake the application up.

 \*/

static void buttons\_leds\_init(bool \* p\_erase\_bonds)

{

 bsp\_event\_t startup\_event;

 uint32\_t err\_code = bsp\_init(BSP\_INIT\_LED | BSP\_INIT\_BUTTONS,

 APP\_TIMER\_TICKS(100, APP\_TIMER\_PRESCALER),

 bsp\_event\_handler);

 APP\_ERROR\_CHECK(err\_code);

 err\_code = bsp\_btn\_ble\_init(NULL, &startup\_event);

 APP\_ERROR\_CHECK(err\_code);

 \*p\_erase\_bonds = (startup\_event == BSP\_EVENT\_CLEAR\_BONDING\_DATA);

}

/\*\*@brief Function for the Power manager.

 \*/

static void power\_manage(void)

{

 uint32\_t err\_code = sd\_app\_evt\_wait();

 APP\_ERROR\_CHECK(err\_code);

}

/\*\*@brief Function for application main entry.

 \*/

int main(void)

{

 uint32\_t err\_code;

 bool erase\_bonds;

 // Initialize.

 timers\_init();

 buttons\_leds\_init(&erase\_bonds);

 ble\_stack\_init();

 device\_manager\_init(erase\_bonds);

 gap\_params\_init();

 advertising\_init();

 services\_init();

 conn\_params\_init();

 // Start execution.

 application\_timers\_start();

 err\_code = ble\_advertising\_start(BLE\_ADV\_MODE\_FAST);

 APP\_ERROR\_CHECK(err\_code);

 // Enter main loop.

 for (;;)

 {

 power\_manage();

 }

}

/\*\*

 \* @}

 \*/