```
  1  /* Copyright (c) 2014 Nordic Semiconductor. All Rights Reserved.
  2   *
  3   * The information contained herein is property of Nordic Semiconductor ASA.
  4   * Terms and conditions of usage are described in detail in NORDIC
  5   * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
  6   *
  7   * Licensees are granted free, non-transferable use of the information. NO
  8   * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
  9   * the file.
 10   *
 11   */
 12
 13  /** @file
 14   *
 15   * @defgroup ble_app_beacon_main main.c
 16   * @{
 17   * @ingroup ble_app_beacon
 18   * @brief Beacon Transmitter Sample Application main file.
 19   *
 20   * This file contains the source code for a beacon transmitter sample application.
 21   */
 22
 23  #include <stdbool.h>
 24  #include <stdint.h>
 25  #include "ble_conn_params.h"
 26  #include "ble.h"
 27  #include "ble_hci.h"
 28  #include "ble_srv_common.h"
 29  #include "ble_advdata.h"
 30  #include "nordic_common.h"
 31  #include "softdevice_handler.h"
 32  #include "pstorage.h"
 33  #include "app_gpiote.h"
 34  #include "app_timer.h"
 35  #include "app_button.h"
 36  #include "pca20006.h"
 37  #include "ble_bcs.h"
 38  #include "ble_dfu.h"
 39  #include "dfu_app_handler_mod.h"
 40  #include "led_softblink.h"
 41  #include "beacon.h"
 42
 43  #include "nrf_sdm.h"    /**FA */
 44  #include "bsp_btn_ble.h"
 45
 46  /* Button definitions */
 47  #define BOOTLOADER_BUTTON_PIN          BUTTON_0                                 /**< Button used to
     enter DFU mode. */
 48  #define CONFIG_MODE_BUTTON_PIN         BUTTON_1                                 /**< Button used to
     enter config mode. */
 49  #define APP_GPIOTE_MAX_USERS           2                                        /**< Max users of
     app_gpiote */
 50  #define BUTTON_DETECTION_DELAY         APP_TIMER_TICKS(50, APP_TIMER_PRESCALER)    /**< Button
     detection delay in ticks */
 51
 52  /* LED definitions */
 53  #define LED_R_MSK                      (1UL << LED_RGB_RED)                        /**< Red LED
     bitmask */
 54  #define LED_G_MSK                      (1UL << LED_RGB_GREEN)                      /**< Green LED
     bitmask */
 55  #define LED_B_MSK                      (1UL << LED_RGB_BLUE)                       /**< Blue LED
     bitmask */
 56  #define APP_CONFIG_MODE_LED_MSK        (LED_R_MSK | LED_G_MSK)                  /**< Blinking
     yellow when device is in config mode */
 57  #define APP_BEACON_MODE_LED_MSK        (LED_R_MSK | LED_B_MSK)                  /**< Blinking
     purple when device is advertising as beacon */
 58
 59  #define ASSERT_LED_PIN_NO              LED_RGB_RED                              /**< Red LED to
     indicate assert */
 60
 61  /* Beacon configuration mode parameters */
 62  #define BEACON_CONFIG_NAME             "BeaconConfig"                           /**< Name of
     device. Will be included in the advertising data. */
```

```
 63
 64  #define APP_CONFIG_ADV_INTERVAL         MSEC_TO_UNITS(100, UNIT_0_625_MS)          /**< The
     advertising interval in configuration mode (in units of 0.625 ms. This value corresponds to 40 ms). */
 65  #define APP_CONFIG_ADV_TIMEOUT          30                                         /**< The
     advertising timeout (in units of seconds). */
 66
 67  #define MIN_CONN_INTERVAL               MSEC_TO_UNITS(500, UNIT_1_25_MS)           /**< Minimum
     acceptable connection interval (0.5 seconds). */
 68  #define MAX_CONN_INTERVAL               MSEC_TO_UNITS(1000, UNIT_1_25_MS)          /**< Maximum
     acceptable connection interval (1 second). */
 69  #define SLAVE_LATENCY                   0                                          /**< Slave latency.
     */
 70  #define CONN_SUP_TIMEOUT                MSEC_TO_UNITS(4000, UNIT_10_MS)            /**< Connection
     supervisory timeout (4 seconds). */
 71
 72  #define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(20000, APP_TIMER_PRESCALER) /**< Time from
     initiating event (connect or start of notification) to first time sd_ble_gap_conn_param_update is
     called (15 seconds). */
 73  #define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(5000, APP_TIMER_PRESCALER)  /**< Time between
     each call to sd_ble_gap_conn_param_update after the first (5 seconds). */
 74  #define MAX_CONN_PARAMS_UPDATE_COUNT    3                                          /**< Number of
     attempts before giving up the connection parameter negotiation. */
 75
 76  /** #define SEC_PARAM_TIMEOUT               30                                         **< Timeout for
     Pairing Request or Security Request (in seconds). */
 77  #define SEC_PARAM_BOND                  0                                          /**< Perform
     bonding. */
 78  #define SEC_PARAM_MITM                  0                                          /**< Man In The
     Middle protection not required. */
 79  #define SEC_PARAM_IO_CAPABILITIES       BLE_GAP_IO_CAPS_NONE                       /**< No I/O
     capabilities. */
 80  #define SEC_PARAM_OOB                   0                                          /**< Out Of Band
     data not available. */
 81  #define SEC_PARAM_MIN_KEY_SIZE          7                                          /**< Minimum
     encryption key size. */
 82  #define SEC_PARAM_MAX_KEY_SIZE          16                                         /**< Maximum
     encryption key size. */
 83
 84  /* App timer parameters */
 85  #define APP_TIMER_PRESCALER             0                                          /**< RTC prescaler
     value used by app_timer */
 86  #define APP_TIMER_MAX_TIMERS            3                                          /**< One for each
     module + one for ble_conn_params + a few extra */
 87  #define APP_TIMER_OP_QUEUE_SIZE         3                                          /**< Maximum number
     of timeout handlers pending execution */
 88
 89  /* App scheduler parameters */
 90  #define SCHED_MAX_EVENT_DATA_SIZE       sizeof(app_timer_event_t)                  /**< Maximum size
     of scheduler events. Note that scheduler BLE stack events do not contain any data, as the events are
     being pulled from the stack in the event handler. */
 91  #define SCHED_QUEUE_SIZE                10                                         /**< Maximum number
     of events in the scheduler queue. */
 92
 93  /* DFU definitions */
 94  #define DFU_REV_MAJOR                   0x00                                       /** DFU Major
     revision number to be exposed. */
 95  #define DFU_REV_MINOR                   0x01                                       /** DFU Minor
     revision number to be exposed. */
 96  #define DFU_REVISION                    ((DFU_REV_MAJOR << 8) | DFU_REV_MINOR)     /** DFU Revision
     number to be exposed. Combined of major and minor versions. */
 97
 98
 99  #define DEAD_BEEF                       0xDEADBEEF                                 /**< Value used as
     error code on stack dump, can be used to identify stack location on stack unwind. */
100  #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*/
101
102
103
104  /* Static variables */
105  static beacon_mode_t        m_beacon_mode;                                         /**< Current beacon
     mode */
```

```
106   static beacon_flash_db_t    *p_beacon;                                  /**< Pointer to
      beacon params */
107   static pstorage_handle_t    m_pstorage_block_id;                        /**< Pstorage
      handle for beacon params */
108
109   static ble_gap_sec_params_t m_sec_params;                               /**< Security
      requirements for this application. */
110   static uint16_t             m_conn_handle = BLE_CONN_HANDLE_INVALID;    /**< Handle of the
      current connection. */
111   static ble_bcs_t            m_bcs;                                      /**< Beacon
      Configuration Service structure.*/
112   static ble_dfu_t            m_dfus;                                     /**< Structure used
      to identify the DFU service. */
113
114   static ble_gap_adv_params_t m_adv_params;                               /**< Parameters to
      be passed to the stack when starting advertising. */
115
116   static ble_gap_scan_params_t m_scan_param;    /**FA */
117   //Address of other device, from Master Control Panel. EC4707F0508D becomes: D6:49:06:FE:AB:22
118   static uint8_t search[BLE_GAP_ADDR_LEN] = {0x22, 0xAB, 0xFE, 0x06, 0x49, 0xD6};
119   static bool   start = false;
120
121   static bool                 m_beacon_reset = false;                     /**< Flag to reset
      system after flash access has finished. */
122   static bool                 m_beacon_start = false;                     /**< Flag to setup
      and start beacon after flash access has finished. */
123
124   /* Prototypes */
125   static void beacon_reset(void);
126   static void beacon_start(beacon_mode_t mode);
127
128   /**@brief Function for error handling, which is called when an error has occurred.
129    *
130    * @warning This handler is an example only and does not fit a final product. You need to analyze
131    *          how your product is supposed to react in case of error.
132    *
133    * @param[in] error_code  Error code supplied to the handler.
134    * @param[in] line_num    Line number where the handler is called.
135    * @param[in] p_file_name Pointer to the file name.
136    */
137   void app_error_handler(uint32_t error_code, uint32_t line_num, const uint8_t * p_file_name)
138   {
139       uint32_t err_code = error_code;
140       uint32_t lin_num  = line_num;
141       const uint8_t * p_fname = p_file_name;
142
143       nrf_gpio_pin_clear(ASSERT_LED_PIN_NO);
144
145       UNUSED_VARIABLE(err_code);
146       UNUSED_VARIABLE(lin_num);
147       UNUSED_VARIABLE(p_fname);
148       // On assert, the system can only recover on reset.
149       NVIC_SystemReset();
150   }
151
152
153   /**@brief Callback function for asserts in the SoftDevice.
154    *
155    * @details This function will be called in case of an assert in the SoftDevice.
156    *
157    * @warning This handler is an example only and does not fit a final product. You need to analyze
158    *          how your product is supposed to react in case of Assert.
159    * @warning On assert from the SoftDevice, the system can only recover on reset.
160    *
161    * @param[in]   line_num   Line number of the failing ASSERT call.
162    * @param[in]   file_name  File name of the failing ASSERT call.
163    */
164   void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
165   {
166       app_error_handler(DEAD_BEEF, line_num, p_file_name);
167   }
168
169   /**@brief Function for handeling storage access complete events.
```

```
170     *
171     * @details This function is called through the sheduler from sys_evt_dispatch when storage access is
        complete
172     *
173     * @param[in]   data   Event data, not used.
174     * @param[in]   size   Event data size, not used.
175     */
176    static void storage_access_complete_handler(void *data, uint16_t size)
177    {
178        if (m_beacon_reset)
179        {
180            beacon_reset();
181        }
182        else if (m_beacon_start)
183        {
184            beacon_start(m_beacon_mode);
185        }
186    }
187
188    /**@brief Function for dispatching a system event to interested modules.
189     *
190     * @details This function is called from the System event interrupt handler after a system
191     *          event has been received.
192     *
193     * @param[in]   sys_evt   System stack event.
194     */
195    static void sys_evt_dispatch(uint32_t sys_evt)
196    {
197        uint32_t err_code;
198        uint32_t count;
199
200        pstorage_sys_event_handler(sys_evt);
201        // Check if storage access is in progress.
202        err_code = pstorage_access_status_get(&count);
203        if ((err_code == NRF_SUCCESS) && (count == 0))
204        {
205            err_code = app_sched_event_put(0, 0, storage_access_complete_handler);
206            APP_ERROR_CHECK(err_code);
207        }
208    }
209
210
211
212    /**@brief Function for the LEDs initialization.
213     *
214     * @details Initializes all LEDs used by this application and starts softblink timer.
215     */
216    static void leds_init(void)
217    {
218        uint32_t err_code;
219        led_sb_init_params_t led_sb_init_params;
220
221        nrf_gpio_cfg_output(LED_RGB_RED);
222        nrf_gpio_cfg_output(LED_RGB_GREEN);
223        nrf_gpio_cfg_output(LED_RGB_BLUE);
224
225        nrf_gpio_pin_set(LED_RGB_RED);
226        nrf_gpio_pin_set(LED_RGB_GREEN);
227        nrf_gpio_pin_set(LED_RGB_BLUE);
228
229        led_sb_init_params.active_high     = false;
230        led_sb_init_params.duty_cycle_max  = 20;
231        led_sb_init_params.duty_cycle_min  = 0;
232        led_sb_init_params.duty_cycle_step = 1;
233        led_sb_init_params.leds_pin_bm     = (LED_R_MSK | LED_G_MSK | LED_B_MSK);
234        led_sb_init_params.off_time_ms     = 4000;
235        led_sb_init_params.on_time_ms      = 0;
236
237        err_code = led_softblink_init(&led_sb_init_params);
238        APP_ERROR_CHECK(err_code);
239    }
240
241    /**@brief Function for initializing the Advertising functionality.
```

```
242     *
243     * @details Encodes the required advertising data and passes it to the stack.
244     *          Also builds a structure to be passed to the stack when starting advertising.
245     */
246    static void advertising_init(beacon_mode_t mode)
247    {
248        if (mode == beacon_mode_normal)
249        {
250            uint32_t        err_code;
251            ble_advdata_t   advdata;
252            uint8_t         flags = BLE_GAP_ADV_FLAG_BR_EDR_NOT_SUPPORTED;
253            ble_advdata_manuf_data_t manuf_specific_data;
254
255            manuf_specific_data.company_identifier = p_beacon->data.company_id;
256            manuf_specific_data.data.p_data        = p_beacon->data.beacon_data;
257            manuf_specific_data.data.size          = APP_BEACON_MANUF_DATA_LEN;
258
259            // Build and set advertising data.
260            memset(&advdata, 0, sizeof(advdata));
261
262            advdata.name_type               = BLE_ADVDATA_NO_NAME;
263            advdata.flags.size              = sizeof(flags);
264            advdata.flags.p_data            = &flags;
265            advdata.p_manuf_specific_data   = &manuf_specific_data;
266
267            err_code = ble_advdata_set(&advdata, NULL);
268            APP_ERROR_CHECK(err_code);
269
270            // Initialize advertising parameters (used when starting advertising).
271            memset(&m_adv_params, 0, sizeof(m_adv_params));
272
273            m_adv_params.type        = BLE_GAP_ADV_TYPE_ADV_NONCONN_IND;
274            m_adv_params.p_peer_addr = NULL;                             // Undirected advertisement.
275            m_adv_params.fp          = BLE_GAP_ADV_FP_ANY;
276            m_adv_params.interval    = MSEC_TO_UNITS(p_beacon->data.adv_interval, UNIT_0_625_MS);
277            m_adv_params.timeout     = APP_BEACON_ADV_TIMEOUT;
278        }
279        else if (mode == beacon_mode_config)
280        {
281            uint32_t      err_code;
282            ble_advdata_t advdata;
283            ble_advdata_t scanrsp;
284            uint8_t       flags = BLE_GAP_ADV_FLAGS_LE_ONLY_LIMITED_DISC_MODE;
285
286            ble_uuid_t adv_uuids[] = {{BCS_UUID_SERVICE, m_bcs.uuid_type}};
287
288            // Build and set advertising data
289            memset(&advdata, 0, sizeof(advdata));
290            advdata.name_type               = BLE_ADVDATA_FULL_NAME;
291            advdata.include_appearance      = true;
292            advdata.flags.size              = sizeof(flags);
293            advdata.flags.p_data            = &flags;
294
295            memset(&scanrsp, 0, sizeof(scanrsp));
296            scanrsp.uuids_complete.uuid_cnt = sizeof(adv_uuids) / sizeof(adv_uuids[0]);
297            scanrsp.uuids_complete.p_uuids  = adv_uuids;
298
299            err_code = ble_advdata_set(&advdata, &scanrsp);
300            APP_ERROR_CHECK(err_code);
301
302            // Initialize advertising parameters (used when starting advertising).
303            memset(&m_adv_params, 0, sizeof(m_adv_params));
304
305            m_adv_params.type        = BLE_GAP_ADV_TYPE_ADV_IND;
306            m_adv_params.p_peer_addr = NULL;
307            m_adv_params.fp          = BLE_GAP_ADV_FP_ANY;
308            m_adv_params.interval    = APP_CONFIG_ADV_INTERVAL;
309            m_adv_params.timeout     = APP_CONFIG_ADV_TIMEOUT;
310        }
311        else
312        {
313            APP_ERROR_CHECK_BOOL(false);
314        }
```

```
315   }
316
317   /**@brief Function for starting advertising.
318    */
319   static void advertising_start(void)
320   {
321       uint32_t err_code;
322
323       err_code = sd_ble_gap_adv_start(&m_adv_params);
324       APP_ERROR_CHECK(err_code);
325   }
326
327   /**@brief Function for the Power manager.
328    */
329   static void power_manage(void)
330   {
331       uint32_t err_code = sd_app_evt_wait();
332       APP_ERROR_CHECK(err_code);
333   }
334
335   /**@brief Function for handeling button presses.
336    */
337   static void button_handler(uint8_t pin_no, uint8_t action)
338   {
339       if (action == APP_BUTTON_PUSH)
340       {
341           if(pin_no == CONFIG_MODE_BUTTON_PIN)
342           {
343               beacon_reset();
344           }
345           else if (pin_no == BOOTLOADER_BUTTON_PIN)
346           {
347               beacon_reset();
348           }
349           else
350           {
351               APP_ERROR_CHECK_BOOL(false);
352           }
353       }
354   }
355
356   /**@brief Function for initializing and enabling the app_button module.
357    */
358   static void buttons_init(void)
359   {
360       uint32_t err_code;
361
362       // @note: Array must be static because a pointer to it will be saved in the Button handler
363       // module.
364       static app_button_cfg_t buttons[] =
365       {
366           {CONFIG_MODE_BUTTON_PIN, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_handler},
367           {BOOTLOADER_BUTTON_PIN, APP_BUTTON_ACTIVE_LOW, BUTTON_PULL, button_handler}
368       };
369
370       APP_BUTTON_INIT(buttons, sizeof(buttons) / sizeof(buttons[0]), BUTTON_DETECTION_DELAY, true);
371
372       err_code = app_button_enable();
373       APP_ERROR_CHECK(err_code);
374   }
375
376
377   /**@brief Function for handling the writes to the configuration characteristics of the beacon
      configuration service.
378    * @detail A pointer to this function is passed to the service in its init structure.
379    */
380   static void beacon_write_handler(ble_bcs_t * p_lbs, beacon_data_type_t type, uint8_t *data)
381   {
382       uint32_t err_code;
383
384       static beacon_flash_db_t tmp;
385
386       memcpy(&tmp, p_beacon, sizeof(beacon_flash_db_t));
```

```
387
388         tmp.data.magic_byte = MAGIC_FLASH_BYTE;
389
390         switch(type)
391         {
392             case beacon_maj_min_data:
393                 tmp.data.beacon_data[BEACON_MANUF_DAT_MAJOR_H_IDX] = data[0];
394                 tmp.data.beacon_data[BEACON_MANUF_DAT_MAJOR_L_IDX] = data[1];
395                 tmp.data.beacon_data[BEACON_MANUF_DAT_MINOR_H_IDX] = data[2];
396                 tmp.data.beacon_data[BEACON_MANUF_DAT_MINOR_L_IDX] = data[3];
397                 break;
398
399             case beacon_measured_rssi_data:
400                 tmp.data.beacon_data[BEACON_MANUF_DAT_RSSI_IDX] = data[0];
401                 break;
402
403             case beacon_uuid_data:
404                 memcpy(&tmp.data.beacon_data[BEACON_MANUF_DAT_UUID_IDX], data, BCS_DATA_ID_LEN);
405                 break;
406
407             case beacon_company_id_data:
408                 tmp.data.company_id = (data[1] << 8) + data[0];
409                 break;
410
411             case beacon_adv_interval_data:
412                 tmp.data.adv_interval = (data[1] << 8) + data[0];
413                 if (tmp.data.adv_interval < APP_BEACON_ADV_INTERVAL_MIN_MS)
414                 {
415                     tmp.data.adv_interval = APP_BEACON_ADV_INTERVAL_MIN_MS;
416                 }
417                 else if (tmp.data.adv_interval > APP_BEACON_ADV_INTERVAL_MAX_MS)
418                 {
419                     tmp.data.adv_interval = APP_BEACON_ADV_INTERVAL_MAX_MS;
420                 }
421                 break;
422
423             case beacon_led_data:
424                 tmp.data.led_state = data[0];
425                 break;
426
427             default:
428                 break;
429         }
430
431         err_code = pstorage_clear(&m_pstorage_block_id, sizeof(beacon_flash_db_t));
432         APP_ERROR_CHECK(err_code);
433
434         err_code = pstorage_store(&m_pstorage_block_id, (uint8_t *)&tmp, sizeof(beacon_flash_db_t), 0);
435         APP_ERROR_CHECK(err_code);
436     }
437
438     /**@brief Function for the GAP initialization.
439      *
440      * @details This function shall be used to setup all the necessary GAP (Generic Access Profile)
441      *          parameters of the device. It also sets the permissions and appearance.
442      */
443     static void gap_params_init(void)
444     {
445         uint32_t                err_code;
446         ble_gap_conn_params_t   gap_conn_params;
447         ble_gap_conn_sec_mode_t sec_mode;
448
449         BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);
450
451         err_code = sd_ble_gap_device_name_set(&sec_mode, (const uint8_t *) BEACON_CONFIG_NAME,
    strlen(BEACON_CONFIG_NAME));
452         APP_ERROR_CHECK(err_code);
453
454         memset(&gap_conn_params, 0, sizeof(gap_conn_params));
455
456         gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
457         gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
458         gap_conn_params.slave_latency     = SLAVE_LATENCY;
```

```
459       gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;
460
461       err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
462       APP_ERROR_CHECK(err_code);
463   }
464
465   /**@brief Function for stopping advertising.
466    */
467   static void advertising_stop(void)
468   {
469       uint32_t err_code;
470
471       err_code = sd_ble_gap_adv_stop();
472       APP_ERROR_CHECK(err_code);
473
474       err_code = led_softblink_stop(APP_CONFIG_MODE_LED_MSK);
475       APP_ERROR_CHECK(err_code);
476   }
477
478   /**@brief Function for preparing before reset.
479    */
480   static void dfu_reset_prepare(void)
481   {
482       uint32_t err_code;
483
484       if (m_conn_handle != BLE_CONN_HANDLE_INVALID)
485       {
486           // Disconnect from peer.
487           err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
488           APP_ERROR_CHECK(err_code);
489           err_code = led_softblink_stop(APP_CONFIG_MODE_LED_MSK);
490           APP_ERROR_CHECK(err_code);
491       }
492       else
493       {
494           // If not connected, then the device will be advertising. Hence stop the advertising.
495           advertising_stop();
496       }
497
498       err_code = ble_conn_params_stop();
499       APP_ERROR_CHECK(err_code);
500   }
501
502   /**@brief Function for initializing services that will be used by the application.
503    */
504   static void services_init(void)
505   {
506       uint32_t err_code;
507       ble_bcs_init_t init;
508       ble_dfu_init_t   dfus_init;
509
510       // Initialize the Device Firmware Update Service.
511       memset(&dfus_init, 0, sizeof(dfus_init));
512
513       dfus_init.evt_handler    = dfu_app_on_dfu_evt;
514       dfus_init.error_handler  = NULL; //service_error_handler - Not used as only the switch from app to
      DFU mode is required and not full dfu service.
515       dfus_init.evt_handler    = dfu_app_on_dfu_evt;
516       dfus_init.revision       = DFU_REVISION;
517
518       err_code = ble_dfu_init(&m_dfus, &dfus_init);
519       APP_ERROR_CHECK(err_code);
520
521       dfu_app_reset_prepare_set(dfu_reset_prepare);
522
523
524       init.beacon_write_handler = beacon_write_handler;
525       init.p_beacon = p_beacon;
526
527       err_code = ble_bcs_init(&m_bcs, &init);
528       APP_ERROR_CHECK(err_code);
529   }
530
```

```
531   /**@brief Function for initializing security parameters.
532    */
533   static void sec_params_init(void)
534   {
535       /** m_sec_params.timeout      = SEC_PARAM_TIMEOUT; */
536       m_sec_params.bond         = SEC_PARAM_BOND;
537       m_sec_params.mitm         = SEC_PARAM_MITM;
538       m_sec_params.io_caps      = SEC_PARAM_IO_CAPABILITIES;
539       m_sec_params.oob          = SEC_PARAM_OOB;
540       m_sec_params.min_key_size = SEC_PARAM_MIN_KEY_SIZE;
541       m_sec_params.max_key_size = SEC_PARAM_MAX_KEY_SIZE;
542   }
543
544   /**@brief Function for handling the Connection Parameters Module.
545    *
546    * @details This function will be called for all events in the Connection Parameters Module which
547    *          are passed to the application.
548    *          @note All this function does is to disconnect. This could have been done by simply
549    *                setting the disconnect_on_fail config parameter, but instead we use the event
550    *                handler mechanism to demonstrate its use.
551    *
552    * @param[in]   p_evt   Event received from the Connection Parameters Module.
553    */
554   static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
555   {
556       uint32_t err_code;
557
558       if(p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
559       {
560           err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
561           APP_ERROR_CHECK(err_code);
562       }
563   }
564
565
566   /**@brief Function for handling a Connection Parameters error.
567    *
568    * @param[in]   nrf_error   Error code containing information about what went wrong.
569    */
570   static void conn_params_error_handler(uint32_t nrf_error)
571   {
572       APP_ERROR_HANDLER(nrf_error);
573   }
574
575
576   /**@brief Function for initializing the Connection Parameters module.
577    */
578   static void conn_params_init(void)
579   {
580       uint32_t               err_code;
581       ble_conn_params_init_t cp_init;
582
583       memset(&cp_init, 0, sizeof(cp_init));
584
585       cp_init.p_conn_params                  = NULL;
586       cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
587       cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
588       cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
589       cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
590       cp_init.disconnect_on_fail             = false;
591       cp_init.evt_handler                    = on_conn_params_evt;
592       cp_init.error_handler                  = conn_params_error_handler;
593
594       err_code = ble_conn_params_init(&cp_init);
595       APP_ERROR_CHECK(err_code);
596   }
597
598   /**@brief Function for handling the Application's BLE Stack events.
599    *
600    * @param[in]   p_ble_evt   Bluetooth stack event.
601    */
602   static void on_ble_evt(ble_evt_t * p_ble_evt)
603   {
```

```
604       uint32_t                          err_code = NRF_SUCCESS;
605       static ble_gap_evt_auth_status_t m_auth_status;
606       ble_gap_enc_info_t *              p_enc_info;
607
608       const ble_gap_evt_t   * p_gap_evt = &p_ble_evt->evt.gap_evt; /**FA */
609
610       switch (p_ble_evt->header.evt_id)
611       {
612         case BLE_GAP_EVT_ADV_REPORT:
613           {
614               if( p_gap_evt->params.adv_report.peer_addr.addr[0] == search[0] && \
615                   p_gap_evt->params.adv_report.peer_addr.addr[1] == search[1] && \
616                   p_gap_evt->params.adv_report.peer_addr.addr[2] == search[2] && \
617                   p_gap_evt->params.adv_report.peer_addr.addr[3] == search[3] && \
618                   p_gap_evt->params.adv_report.peer_addr.addr[4] == search[4] && \
619                   p_gap_evt->params.adv_report.peer_addr.addr[5] == search[5])
620               {
621                   nrf_gpio_pin_toggle(LED_RGB_BLUE);
622                   start = true;
623               }
624           }
625
626         case BLE_GAP_EVT_CONNECTED:
627               m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
628               break;
629
630           case BLE_GAP_EVT_DISCONNECTED:
631               m_conn_handle = BLE_CONN_HANDLE_INVALID;
632               beacon_reset();
633               break;
634
635           case BLE_GAP_EVT_SEC_PARAMS_REQUEST:
636               err_code = sd_ble_gap_sec_params_reply(m_conn_handle,
637                                                      BLE_GAP_SEC_STATUS_SUCCESS,
638                                                      &m_sec_params, NULL);
639               break;
640
641           case BLE_GATTS_EVT_SYS_ATTR_MISSING:
642               err_code = sd_ble_gatts_sys_attr_set(m_conn_handle, NULL, 0, NULL);
643               break;
644
645           case BLE_GAP_EVT_AUTH_STATUS:
646               m_auth_status = p_ble_evt->evt.gap_evt.params.auth_status;
647               break;
648
649   //        case BLE_GAP_EVT_SEC_INFO_REQUEST:
650   //            p_enc_info = &m_auth_status.periph_keys.enc_info;
651   //            if (p_enc_info->div == p_ble_evt->evt.gap_evt.params.sec_info_request.div)
652   //            {
653   //                err_code = sd_ble_gap_sec_info_reply(m_conn_handle, p_enc_info, NULL);
654   //            }
655   //            else
656   //            {
657   //                // No keys found for this device
658   //                err_code = sd_ble_gap_sec_info_reply(m_conn_handle, NULL, NULL);
659   //            }
660   //            break;
661
662           case BLE_GAP_EVT_TIMEOUT:
663               if (p_ble_evt->evt.gap_evt.params.timeout.src == BLE_GAP_TIMEOUT_SRC_ADVERTISING)
664               {
665                   beacon_reset();
666               }
667               break;
668
669           default:
670               break;
671       }
672
673       APP_ERROR_CHECK(err_code);
674   }
675
676   /**@brief Function for dispatching a BLE stack event to all modules with a BLE stack event handler.
```

```
677     *
678     * @details This function is called from the scheduler in the main loop after a BLE stack
679     *          event has been received.
680     *
681     * @param[in]   p_ble_evt   Bluetooth stack event.
682     */
683    static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
684    {
685        ble_conn_params_on_ble_evt(p_ble_evt);
686        ble_bcs_on_ble_evt(&m_bcs, p_ble_evt);
687        ble_dfu_on_ble_evt(&m_dfus, p_ble_evt);
688        on_ble_evt(p_ble_evt);
689    }
690
691    /**@brief Function for initializing the BLE stack.
692     *
693     * @details Initializes the SoftDevice and the BLE event interrupt.
694     */
695    static void ble_stack_init(void)
696    {
697        uint32_t err_code;
698
699        // Initialize the SoftDevice handler module.
700        SOFTDEVICE_HANDLER_INIT(NRF_CLOCK_LFCLKSRC_XTAL_50_PPM, true);
701
702        // Enable BLE stack
703        ble_enable_params_t ble_enable_params;
704        memset(&ble_enable_params, 0, sizeof(ble_enable_params));
705
706    #ifdef S130
707        ble_enable_params.gatts_enable_params.attr_tab_size   = BLE_GATTS_ATTR_TAB_SIZE_DEFAULT;
708    #endif
709
710        ble_enable_params.gatts_enable_params.service_changed = IS_SRVC_CHANGED_CHARACT_PRESENT;
711        err_code = sd_ble_enable(&ble_enable_params);
712        APP_ERROR_CHECK(err_code);
713
714        // Register with the SoftDevice handler module for BLE events.
715        err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
716        APP_ERROR_CHECK(err_code);
717
718        // Register with the SoftDevice handler module for BLE events.
719        err_code = softdevice_sys_evt_handler_set(sys_evt_dispatch);
720        APP_ERROR_CHECK(err_code);
721    }
722
723    /** @brief Function for handling pstorage events
724     */
725    static void pstorage_ntf_cb(pstorage_handle_t *  p_handle,
726                                uint8_t              op_code,
727                                uint32_t             result,
728                                uint8_t *            p_data,
729                                uint32_t             data_len)
730    {
731        APP_ERROR_CHECK(result);
732    }
733
734    /** @brief Function for initializing pstorage
735     */
736    static void flash_access_init(void)
737    {
738        uint32_t err_code;
739
740        err_code = pstorage_init();
741        APP_ERROR_CHECK(err_code);
742    }
743
744    /** @brief Function to get a pointer to beacon parameters in flash. Should only be called once during
        initialization.
745    */
746    static beacon_flash_db_t * beacon_params_get(void)
747    {
748        uint32_t err_code;
```

```
749       pstorage_module_param_t pstorage_param;
750
751       pstorage_param.cb = pstorage_ntf_cb;
752       pstorage_param.block_size = sizeof(beacon_flash_db_t);
753       pstorage_param.block_count = 1;
754
755       err_code = pstorage_register(&pstorage_param, &m_pstorage_block_id);
756       APP_ERROR_CHECK(err_code);
757
758       return (beacon_flash_db_t *)m_pstorage_block_id.block_id;
759   }
760
761   /** @brief Function for writing default beacon parameters to flash.
762   */
763   static void beacon_params_default_set(void)
764   {
765       uint32_t err_code;
766       static beacon_flash_db_t tmp;
767
768       uint8_t beacon_data[APP_BEACON_MANUF_DATA_LEN] =  /**< Information advertised by the beacon. */
769       {
770           APP_DEVICE_TYPE,           // Manufacturer specific information. Specifies the device type in
      this
771                                      // implementation.
772           APP_ADV_DATA_LENGTH,       // Manufacturer specific information. Specifies the length of the
773                                      // manufacturer specific data in this implementation.
774           APP_DEFAULT_BEACON_UUID,   // 128 bit UUID value.
775           APP_DEFAULT_MAJOR_VALUE,   // Major arbitrary value that can be used to distinguish between
      beacons.
776           APP_DEFAULT_MINOR_VALUE,   // Minor arbitrary value that can be used to distinguish between
      beacons.
777           APP_DEFAULT_MEASURED_RSSI  // Manufacturer specific information. The beacon's measured TX power
      in
778                                      // this implementation.
779       };
780
781       tmp.data.magic_byte   = MAGIC_FLASH_BYTE;
782       tmp.data.adv_interval = APP_BEACON_DEFAULT_ADV_INTERVAL_MS;
783       tmp.data.company_id   = APP_DEFAULT_COMPANY_IDENTIFIER;
784       tmp.data.led_state    = 0x01;
785
786       beacon_data[BEACON_MANUF_DAT_MINOR_L_IDX] = (uint8_t)(NRF_FICR->DEVICEADDR[0] & 0xFFUL);
787       beacon_data[BEACON_MANUF_DAT_MINOR_H_IDX] = (uint8_t)((NRF_FICR->DEVICEADDR[0] >>  8) & 0xFFUL);
788       beacon_data[BEACON_MANUF_DAT_MAJOR_L_IDX] = (uint8_t)((NRF_FICR->DEVICEADDR[0] >> 16) & 0xFFUL);
789       beacon_data[BEACON_MANUF_DAT_MAJOR_H_IDX] = (uint8_t)((NRF_FICR->DEVICEADDR[0] >> 24) & 0xFFUL);
790
791       memcpy(tmp.data.beacon_data, beacon_data, APP_BEACON_MANUF_DATA_LEN);
792
793       err_code = pstorage_clear(&m_pstorage_block_id, sizeof(beacon_flash_db_t));
794       APP_ERROR_CHECK(err_code);
795
796       err_code = pstorage_store(&m_pstorage_block_id, (uint8_t *)&tmp, sizeof(beacon_flash_db_t), 0);
797       APP_ERROR_CHECK(err_code);
798   }
799
800   /** @brief Function for setup of beacon operation.
801   */
802   static void beacon_setup(beacon_mode_t mode)
803   {
804       if(mode == beacon_mode_config)
805       {
806           gap_params_init();
807           services_init();
808           advertising_init(mode);
809           conn_params_init();
810           sec_params_init();
811           led_softblink_off_time_set(2000);
812           led_softblink_start(APP_CONFIG_MODE_LED_MSK);
813       }
814       else
815       {
816           advertising_init(mode);
817
```

```
818             if (p_beacon->data.led_state)
819             {
820                 led_softblink_start(APP_BEACON_MODE_LED_MSK);
821             }
822         }
823     }
824
825     /** @brief Function for reseting the beacon.
826     */
827     static void beacon_reset(void)
828     {
829         uint32_t err_code;
830         uint32_t count;
831
832         // Check if storage access is in progress.
833         err_code = pstorage_access_status_get(&count);
834         APP_ERROR_CHECK(err_code);
835         if (count == 0)
836         {
837             m_beacon_reset = false;
838             NVIC_SystemReset();
839         }
840         else
841         {
842             m_beacon_reset = true;
843         }
844     }
845
846     /** @brief Function for starting beacon operation.
847     */
848     static void beacon_start(beacon_mode_t mode)
849     {
850         uint32_t err_code;
851         uint32_t count;
852
853         // Check if storage access is in progress.
854         err_code = pstorage_access_status_get(&count);
855         APP_ERROR_CHECK(err_code);
856         if (count == 0)
857         {
858             m_beacon_start = false;
859             // Setup beacon mode
860             beacon_setup(m_beacon_mode);
861
862             // Start advertising
863             advertising_start();
864         }
865         else
866         {
867             m_beacon_start = true;
868         }
869     }
870
871     /** @brief Function for reading the beacon mode button.
872     */
873     static beacon_mode_t beacon_mode_button_read(void)
874     {
875         uint32_t err_code;
876         bool config_mode;
877
878         err_code = app_button_is_pushed(0, &config_mode);
879         APP_ERROR_CHECK(err_code);
880
881         return config_mode ? beacon_mode_config : beacon_mode_normal;
882     }
883
884     /**
885      * @brief Function for application main entry.
886      */
887     static void scan_init(void)
888     {
889         m_scan_param.active = 0;
890         m_scan_param.selective = 0;
```

```
891         m_scan_param.p_whitelist = NULL;
892         m_scan_param.timeout = 0;
893         m_scan_param.interval = 0x00A0;   // 100 ms
894         m_scan_param.window = 0x009C;
895     }
896 
897     static void scan_start(void)
898     {
899         uint32_t err_code;
900         err_code = sd_ble_gap_scan_start(&m_scan_param);
901         APP_ERROR_CHECK(err_code);
902     }
903 
904     int main(void)
905     {
906         // Initialize.
907         APP_SCHED_INIT(SCHED_MAX_EVENT_DATA_SIZE, SCHED_QUEUE_SIZE);
908         APP_TIMER_INIT(APP_TIMER_PRESCALER, APP_TIMER_MAX_TIMERS, APP_TIMER_OP_QUEUE_SIZE, false);
909         APP_GPIOTE_INIT(APP_GPIOTE_MAX_USERS);
910         buttons_init();
911         leds_init();
912         ble_stack_init();
913         flash_access_init();
914 
915         scan_init();
916         scan_start();
917         //led_softblink_off_time_set(2000);
918         led_softblink_start(LED_R_MSK);
919         while (!start)
920         {
921 
922         }
923 
924 
925         // Read beacon mode
926         m_beacon_mode = beacon_mode_button_read();
927         // Read beacon params from flash
928         p_beacon = beacon_params_get();
929         if (p_beacon->data.magic_byte != MAGIC_FLASH_BYTE)
930         {
931             // No valid params found, write default params.
932             beacon_params_default_set();
933         }
934 
935         beacon_start(m_beacon_mode);
936 
937         // Enter main loop.
938         for (;;)
939         {
940             app_sched_execute();
941             power_manage();
942         }
943     }
944 
945     /**
946      * @}
947      */
948 
```