s13x_nrf5x migration document

Introduction to the s13x_nrf5x migration document

About the document

This document describes how to migrate to new versions of the s130_nrf51 and s132_nrf52 SoftDevices. The s130_nrf51 and s132_nrf52 release notes should be read in conjunction with this document.

For each version, we have the following sections:

- "Required changes" describes how an application would have used the previous version of the SoftDevice, and how it must now use this version for the given change.
- "New functionality" describes how to use new features and functionality offered by this version of the SoftDevice. **Note:** Not all new functionality may be covered; the release notes will contain a full list of new features and functionality.

Each section describes how to migrate to a given version from the previous version. If you are migrating to the current version from the previous version, follow the instructions in that section. To migrate between versions that are more than one version apart, follow the migration steps for all intermediate versions in order.

Example: To migrate from version 5.0.0 to version 5.2.0, first follow the instructions to migrate to 5.1.0 from 5.0.0, then follow the instructions to migrate to 5.2.0 from 5.1.0.

s132_nrf52_3.0.0

This section describes how to migrate to s132_nrf52_3.0.0 from s132_nrf52_2.0.1.

Required changes

SoftDevice flash and RAM usage

The size of the SoftDevice has changed requiring a change to the application project file.

For Keil this means:

- Go into the properties of the project and find the Target tab
- Change IROM1 Start to `0x1F000`.

If the project uses a scatter file or linker script instead, those must be updated accordingly.

The RAM usage of SoftDevice has also changed. `sd_ble_enable()` should be used to find the APP_RAM_BASE for a particular configuration.

LL Privacy

This SoftDevice brings in support for LL Privacy. All applications must be updated to the new Privacy API and whitelist API supporting this new feature. Refer to the description of LL privacy in the New functionality section for more details.

Required changes:

- **Enable privacy**

```
/* S132 v2.0 API usage */

ble_gap_addr_t private_addr = {0};
private_addr.addr_type = BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE;
sd_ble_gap_addr_set(BLE_GAP_ADDR_CYCLE_MODE_AUTO, private_addr);
```

```
/* S132 v3.0 API usage */

ble_gap_privacy_params_t privacy_params = {0};
privacy_params.privacy_mode = BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY;
privacy_params.private_addr_type =
BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE;
sd_ble_gap_privacy_set(privacy_params);
```

- **Disable privacy**

```
/* S132 v2.0 API usage */

ble_gap_addr_t identity_addr = saved_identity_addr; /* From
sd_ble_gap_addr_get(). */
sd_ble_gap_addr_set(BLE_GAP_ADDR_CYCLE_MODE_NONE, identity_addr);
```

```
/* S132 v3.0 API usage */

ble_gap_privacy_params_t privacy_params = {0};
privacy_params.privacy_mode = BLE_GAP_PRIVACY_MODE_OFF;
sd_ble_gap_privacy_set(privacy_params);
```

- **Whitelist private addresses**

```
/* S132 v2.0 API usage */

/* Public devices. */
ble_gap_addr_t public_device1 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06}};
ble_gap_addr_t public_device2 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0x10, 0x20, 0x30, 0x40, 0x50, 0x60}};

/* IRKs of Private devices. */
ble_gap_irk_t irk1 = { .irk = { 0x10, 0x20, 0x30 /*...*/} };
ble_gap_irk_tt irk2 = { .irk = { 0x01, 0x02, 0x03 /*...*/} };

ble_gap_addr_t * whitelist_addrs[2] = {&public_device1, &public_device2};
ble_gap_irk_t * whitelist_irks[2] = {&irk1, &irk2};
ble_gap_whitelist_t whitelist = {
  .pp_addrs = &whitelist_addrs, .addr_count = 2, /* Public devices. */
  .pp_irks = &whitelist_irks, .irk_count = 2, /* Private devices. */};

ble_gap_adv_params_t adv_params = {0};
adv_params.p_whitelist = &whitelist
sd_ble_gap_adv_start(&adv_params);
```

```
/* S132 v3.0 API usage */

ble_gap_addr_t public_device1 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
};
ble_gap_addr_t public_device2 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0x10, 0x20, 0x30, 0x40, 0x50, 0x60},
};
  /* Private devices. Matches addresses in identity list. */
ble_gap_addr_t private_device1 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6}
};
ble_gap_addr_t private_device2 = {
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .addr = {0x1A, 0x2A, 0x3A, 0x4A, 0x5A, 0x6A},
};
ble_gap_addr_t * whitelist[4] = {
  &public_device1, &public_device2,
  &private_device1, &private_device2,
};
ble_gap_id_key_t identity1 = {
  .id_addr_info = {
    .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
    .addr = {0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6},},
  .id_info ={
      .irk = { 0x10, 0x20, 0x30 /*...*/},}
};
ble_gap_id_key_t identity2 = {
  .id_addr_info = {
    .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
    .addr = {0x1A, 0x2A, 0x3A, 0x4A, 0x5A, 0x6A},},
  .id_info = {
        .irk = { 0x01, 0x02, 0x03 /*...*/},}
};

ble_gap_id_key_t * identities[2] = { &identity1, &identity2 };
sd_ble_gap_device_identities_set(&identities, NULL /* Don't use local IRKs*/,
2);
sd_ble_gap_whitelist_set(&whitelist, 4);
ble_gap_adv_params_t adv_params = {0};
adv_params.fp = BLE_GAP_ADV_FP_FILTER_BOTH;
sd_ble_gap_adv_start(&adv_params);
```

- **Private address information returned in BLE events**

```
/* S132 v2.0 API usage */

/* GAP connection parameter */
ble_gap_evt_connected_t conn_evt;
conn_evt.irk_match; /* Set to true if IRK matched. */
conn_evt.irk_match_idx; /* Set to index into pp_irks in whitelist.*/
conn_evt.peer_addr; /* Set to the private resolvable address of the peer.*/
```

```
/* S132 v3.0 API usage */

/* ble_gap_addr_t has been updated.
The events ble_gap_evt_connected_t, ble_gap_evt_adv_report_t
and ble_gap_evt_scan_req_report_t are affected. */
ble_gap_addr_t.addr_id_peer; /* Set to true if IRK matched  */
ble_gap_addr_t.addr; /* Set to the identity address of the peer,
                       i.e the one in the identity list matching the
                       peer IRK.*/
```

- **Central connection to peers using private address**

```
/* S132 v2.0 API usage */

/* IRK of the Private device. */
ble_gap_irk_t irk1 = { .irk = { 0x10, 0x20, 0x30 /*...*/} };
ble_gap_irk_t * whitelist_irk[1] = {&irk1};
ble_gap_whitelist_t whitelist = {
  .pp_irks = &whitelist_irk, .irk_count = 1,};

ble_gap_scan_params_t scan_params = {
.selective = true, p_whitelist = &whitelist};
sd_ble_gap_connect(NULL, &scan_params, &conn_params);
```

```
/* S132 v3.0 API usage */

ble_gap_addr_t peer_addr = {
  .addr_id_peer = 1;
  .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC;
  .addr = {0x1A, 0x2A, 0x3A, 0x4A, 0x5A, 0x6A};
}
sd_ble_gap_connect(&peer_addr, &scan_params, &conn_params);
```

LE Ping

The LE ping feature is now supported by the SoftDevice. A new timeout source `BLE_GAP_TIMEOUT_SRC_AUTH_PAYLOAD` has been added. All applications must handle this event from the SoftDevice according to the API documentation. Refer to the description of LE Ping in the New functionality section for more details.

Required changes:

```
/* S132 v3.0 API usage */

/* Ignore the authenticated payload timeout event */
case BLE_GAP_TIMEOUT_SRC_AUTH_PAYLOAD:
  break;
```

Configurable ATT_MTU

The feature of configurable ATT_MTU is now supported by the SoftDevice. A new event `BLE_GATTS_EVT_EXCHANGE_MTU_REQUEST` has been added. All applications must handle this event from the SoftDevice according to the API documentation. Refer to the description of configurable ATT_MTU in the New functionality section for more details.

Required changes:

```
/* S132 v3.0 API usage */

/* Respond with default ATT_MTU, if peer initiates an ATT_MTU exchange procedure.
*/
case BLE_GATTS_EVT_EXCHANGE_MTU_REQUEST:
 sd_ble_gatts_exchange_mtu_reply(p_ble_evt->evt.gatts_evt.conn_handle,
GATT_MTU_SIZE_DEFAULT);
 break;
```

New functionality

Configurable ATT_MTU

The Configurable ATT_MTU feature enables the ATT protocol to use packets longer than the default of 23 bytes. This can be useful for example to reduce complexity of GATTC and GATTS procedures used to handle longer Characteristic Value, where a single "Write Request" can be used instead of the whole "Queued Writes" procedure.

API updates

- A new BLE initialization structure, `ble_gatt_enable_params_t`, has been added to `ble_enable_params_t` for configuring the maximum ATT_MTU the SoftDevice can send or receive.
- A new SV call, `sd_ble_gattc_exchange_mtu_request()`, has been added for starting an ATT_MTU exchange.
- A new SV call, `sd_ble_gatts_exchange_mtu_reply()`, has been added for setting the ATT_MTU in ATT_MTU response.
- A new event, `BLE_GATTS_EVT_EXCHANGE_MTU_REQUEST`, has been added to `BLE_GATTS_EVTS` to notify that an ATT_MTU request has been received. `sd_ble_gatts_exchange_mtu_reply()` must be called by the application, after this event has been received, to continue the ATT_MTU exchange procedure.
- A new event, `BLE_GATTC_EVT_EXCHANGE_MTU_RSP`, has been added to `BLE_GATTC_EVTS` to notify that an ATT_MTU response has been received. This event marks the end of the ATT_MTU exchange procedure and indicates the server ATT_MTU.

Usage

- ATT_MTU exchange can be initiated locally or by peer device.
- HVx and service changed cannot run while a local client initiated ATT_MTU exchange is active. The SV calls `sd_ble_gatts_hvx()` and `sd_ble_gatts_service_changed()` will return NRF_ERROR_INVALID_STATE if a local client initiated ATT_MTU exchange is ongoing.
- Following is the pseudo code for case where ATT_MTU exchange is initiated by application:

```
ble_enable_params_t enable_params = {0};

/* Set maximum ATT_MTU the SoftDevice can send or receive */
enable_params.gatt_enable_params.att_mtu = 158;

/* Set other BLE Initialization parameters */

/* Enable the BLE Stack */
sd_ble_enable(&enable_params, ... );

[...]

uint16_t conn_handle;
/* Establish connection */

[...]

/* Start ATT_MTU exchange */
sd_ble_gattc_exchange_mtu_request(conn_handle, client_rx_mtu);

[...]

uint16_t effective_att_mtu;
uint16_t server_rx_mtu;
/* Handle the event */
case BLE_GATTC_EVT_EXCHANGE_MTU_RSP:
  server_rx_mtu = p_ble_evt->evt.gattc_evt.params.exchange_mtu_rsp.server_rx_mtu;

  /* New ATT_MTU is now applied to GATT procedures for this connection */
  /*Note
   The SoftDevice sets ATT_MTU to the minimum of:
           - The Client RX MTU value from BLE_GATTS_EVT_EXCHANGE_MTU_REQUEST, and
           - The Server RX MTU value.

           However, the SoftDevice never sets ATT_MTU lower than
GATT_MTU_SIZE_DEFAULT.
  */
  /* Store ATT_MTU for later use */
  effective_att_mtu = MIN( MAX(GATT_MTU_SIZE_DEFAULT, server_rx_mtu)
                           , client_rx_mtu
                          );
```

LE Ping

The LE Ping feature can be used by the application to configure a link to try to have at least one authenticated packet exchange within a configurable timeout period. If the peer device does not send an authenticated packet within the timeout, a timeout event is generated to notify this to the application.

API updates

- A new GAP option, `BLE_GAP_OPT_AUTH_PAYLOAD_TIMEOUT`, has been added to set the authenticated payload timeout.
- A new GAP timeout source, `BLE_GAP_TIMEOUT_SRC_AUTH_PAYLOAD`, has been added to indicate that the authenticated payload timer has expired.

Usage

```
/* Enable the BLE Stack */

[...]

/* Establish connection */

[...]

/* Authenticated payload timer runs with default value.
Set the authenticated payload timeout for the link, if required to be something
else then the default */
gap_opt.auth_payload_timeout.conn_handle = connection_handle;
gap_opt.auth_payload_timeout.auth_payload_timeout = 1000;
gap_opt_set(BLE_GAP_OPT_AUTH_PAYLOAD_TIMEOUT, &gap_opt);

[...]

/* Handle the event */
case BLE_GAP_TIMEOUT_SRC_AUTH_PAYLOAD:
 /* Handling of the event is application dependent. It can be ignored if not used
by application. */
 break;
```

LE Data Packet Length Extension (DLE)

The LE Data Packet Length Extension feature enables the SoftDevice to use longer packets on the link layer level. Now link layer packets with up to 251 bytes payload are supported.

API updates

- A new GAP option, `BLE_GAP_OPT_EXT_LEN`, has been added to set the maximum Link Layer PDU length to be used in DLE.
- A new event, `BLE_EVT_DATA_LENGTH_CHANGED`, has been added to indicate that the Link Layer PDU length has changed.

Usage

- Default max Link Layer PDU is 27 bytes.
- `BLE_GAP_OPT_EXT_LEN` changes the max length for all future links.
- Example pseudo code:

```
/* Enable the BLE Stack */

[...]

/* Set max Link Layer PDU length, if application wants it to be more than
27bytes */
gap_opt.ext_len.rxtx_max_pdu_payload_size = 54; //Example: set max length to
54bytes
gap_opt_set(BLE_GAP_OPT_EXT_LEN, &gap_opt);

[...]

/* Establish connection */

[...]

/* Handle the event */
case BLE_EVT_DATA_LENGTH_CHANGED:
  /* Handling of the event is application dependent. It can be ignored if not
used by application. */
```

LL Privacy

The LL Privacy feature provides similar functionality as the privacy in the previous version of the SoftDevice. In addition, it supports new use cases like enabling privacy for directed advertising and advanced filter policy for scanning.

API updates

- New SV calls, `sd_ble_gap_privacy_set()` and `sd_ble_gap_privacy_get()`, are added to set and get the privacy settings. `ble_gap_privacy_params_t` is defined to be used with these calls.
- The GAP option `BLE_GAP_OPT_PRIVACY` is removed. The SV calls `sd_ble_gap_privacy_set()` and `sd_ble_gap_privacy_get()` should be used instead.
- A new GAP characteristic, `BLE_UUID_GAP_CHARACTERISTIC_CAR`, has been added for Central Address Resolution.
- The SV calls `sd_ble_gap_address_set()` and `sd_ble_gap_address_get()` have been renamed to `sd_ble_gap_addr_set()` and `sd_ble_gap_addr_get()` respectively.
- A new SV call, `sd_ble_gap_whitelist_set()`, has been added to set the whitelist. The configured whitelist is shared among all BLE roles.
- A new SV call, `sd_ble_gap_device_identities_set()`, has been added to set the identity list. The configured identity list is shared among all BLE roles.
- New definitions, `BLE_GAP_PRIVACY_MODE_OFF` and `BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY`, have been added.
- Two new GAP error codes, `BLE_ERROR_GAP_DEVICE_IDENTITIES_IN_USE` and `BLE_ERROR_GAP_DEVICE_IDENTITIES_DUPLICATE`, have been added.
- Address cycling, `BLE_GAP_ADDR_CYCLE_MODE_NONE` and `BLE_GAP_ADDR_CYCLE_MODE_AUTO`, is removed from GAP API `sd_ble_gap_addr_set()`. Address will always cycle if privacy is enabled by `sd_ble_gap_privacy_set()`.
- New definitions, `BLE_GAP_DEFAULT_PRIVATE_ADDR_CYCLE_INTERVAL_S` and `BLE_GAP_MAX_PRIVATE_ADDR_CYCLE_INTERVAL_S`, have been added for address cycle intervals.
- `BLE_GAP_WHITELIST_IRK_MAX_COUNT` is renamed to `BLE_GAP_DEVICE_IDENTITIES_MAX_COUNT`.
- A new field, `addr_id_peer`, has been added in the `ble_gap_addr_type_t`, which indicates an IRK/identity match of a peer.
- `ble_gap_whitelist_t` is removed because it is not used anymore. This also means that it is removed from `ble_gap_adv_params_t` and `ble_gap_scan_params_t`. `sd_ble_gap_whitelist_set()` is supposed to be used instead for setting the whitelist.
- `ble_gap_scan_params_t` is updated. "`adv_dir_report`" field has been added to enable extended scanner filter policies.
- `ble_gap_evt_connected_t` is updated. "`own address`", "`irk_match`" and "`irk_match_index`" fields are removed. "`irk_match`" is now given by "`addr_id_peer`" fileld in "`peer_addr`".
- `ble_gap_evt_adv_report_t` is updated and a new field, "`direct_addr`", has been added to support extended scanner filter policy.

Usage

- Example pseudo code using the new privacy API:

```
/* Enable the BLE Stack */

[...]

/* Enable privacy */
ble_gap_privacy_params_t privacy_params = {0};
privacy_params.privacy_mode = BLE_GAP_PRIVACY_MODE_DEVICE_PRIVACY;
privacy_params.private_addr_type =
BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE;
privacy_params.private_addr_cycle_s = 0; /* Default cycle period will be used.
*/
privacy_params.p_device_irk = &own_irk;
sd_ble_gap_privacy_set(&privacy_params);

[...]

/* start scanner and get adv_report */

[...]

/* Connect to chosen advertiser(advertiser using private address). */
ble_gap_addr_t peer_addr = {
  .addr_id_peer = 0;
  .addr_type = BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE;
  .addr = {0xCC, 0xBB, 0xAA, 0xAA, 0xBB, 0xCC};
}
sd_ble_gap_connect(&peer_addr, &scan_params, &conn_params);

[...]

/* Perform bonding */

[...]

/* With IRK exchanged, the identity list can be configured to enable address
resolution.*/
ble_gap_id_key_t identity = {
  .id_addr_info = {
    .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
    .addr = {0x1A, 0x2A, 0x3A, 0x4A, 0x5A, 0x6A},},
  .id_info = {
        .irk = { 0x01, 0x02, 0x03 /*...*/},}
};
ble_gap_id_key_t * identities[] = { &identity };
sd_ble_gap_identities_set(&identities, NULL, 1);

[...]

/* For all future connections, IRK filtering will be performed. */
ble_gap_addr_t peer_addr = {
  .addr_id_peer = 1;
    .addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
    .addr = {0x1A, 0x2A, 0x3A, 0x4A, 0x5A, 0x6A}
}
sd_ble_gap_connect(&peer_addr, &scan_params, &conn_params);
```

```
[...]

/* It is also possible to use extended filter policy to perform IRK resolution
on directed adv reports. */
ble_gap_scan_params_t scan_params;
scan_params.adv_dir_report = 1;
sd_ble_gap_scan_start(&scan_params);

[...]

/* Handle the event */
case BLE_GAP_EVT_ADV_REPORT:
 /* Adv report will also be generated for directed advertisements where
 the initiator field is set to a private resolvable address, even if
 the address did not resolve to an entry in the device identity list.*/
 if (ble_evt->adv_report.type == BLE_GAP_ADV_TYPE_ADV_DIRECT_IND)
 {
   if (ble_evt->adv_report.direct_addr.addr_type ==
       BLE_GAP_ADDR_TYPE_RANDOM_PRIVATE_RESOLVABLE)
   {
     // The initiator address is not resolved
   }
   else
   {
     // The initiator address is resolved
```

```
    }
  }
```

Connection Event Length Extension

This feature can be used to dynamically extend the connection event length when possible to send extra packets compared to the configured bandwidth in a connection interval.

API updates

- A new option, `BLE_COMMON_OPT_CONN_EVT_EXT`, has been added to `BLE_COMMON_OPTS` for enabling/disabling of this feature.

Usage

- This feature of dynamic extension of connection event length is disabled by default.
- The `BLE_COMMON_OPT_CONN_EVT_EXT` option can be used to enable/disable this feature. This will result in enabling/disabling this feature for all currently active links and also for all future links.

Full length device name

The maximum possible length of the device name has been increased, and it is now possible to set a device name up to 248 bytes.

API updates

- A new parameter, `ble_gap_device_name_t`, has been added to `sd_ble_enable()` for setting full length device name.

Usage

- Example pseudo code:

```
ble_enable_params_t enable_params = {0};

/* Set the device name, if application wants to set anything longer than
BLE_GAP_DEVNAME_DEFAULT_LEN */
ble_gap_device_name_t device_name = {0};
uint8_t device_name_buff[BLE_GAP_DEVNAME_MAX_LEN] = "My very long exciting
application name";
device_name.vloc = BLE_GATTS_VLOC_STACK; /*Note: Device name will occupy space
in Attribute Table.*/
device_name.p_value = device_name_buff;
device_name.max_len = sizeof(device_name_buff);
device_name.current_len = strlen((char *)device_name_buff);
enable_params.gap_enable_params.p_device_name = &device_name;

/* Set other BLE Initialization parameters */
sd_ble_enable(&enable_params, ... );

[...]
```

Max BLE event length calculation

The maximum size of a BLE event can now be calculated to optimize the size of event buffer memory.

API updates

- A new macro, `BLE_EVTS_LEN_MAX`, has been added to find out the maximum size of BLE events.

Usage

```
/* Old API: */

uint8_t evt[sizeof(ble_evt_t) + BLE_L2CAP_MTU_DEF];
uint16_t evt_len = sizeof(evt);

errcode = sd_ble_evt_get(evt, &evt_len);
```

```
/* New API: */

uint8_t evt[BLE_EVTS_LEN_MAX(GATT_MTU_SIZE_DEFAULT)];
uint16_t evt_len = sizeof(evt);

errcode = sd_ble_evt_get(evt, &evt_len);
```

Other miscellaneous updates

- The SoftDevice Information Structure has been updated and new access macros have been added. Note that this these updates are for Nordic internal use and should not be used by the application.
- New access macros for general purpose retention registers have been added.

API diff

A diff of the API changes between versions s132_nrf52_3.0.0 and s132_nrf5x_2.0.1 is provided with this release. Refer to the file s132_nrf52_3.0.0_API-update.diff.

s13x_nrf5x_2.0.1

This section describes how to migrate to s13x_nrf5x_2.0.1 from s130_nrf51_1.0.0.

Required changes

SoftDevice size

The size of the SoftDevice has changed requiring a change to the application project file.

For Keil this means:

- Go into the properties of the project and find the Target tab
- Change IROM1 Start to **`0x1B000`** (s130) or **`0x1C000`** (s132).

If the project uses a scatter file or linker script instead, those must be updated accordingly.

SVC number changes

The SVC numbers in use by the SoftDevice have been changed so the application needs to be recompiled against the new header files.

Fault handling

The SoftDevice has changed the way it handles unrecoverable errors, now known as "faults". SoftDevice assertions were reported to the application in previous releases, now a wider range of faults has been introduced and a new handling mechanism. The new format for the fault handler to be supplied to `sd_softdevice_enable()` reflects this.

The old

`typedef void (*softdevice_assertion_handler_t)(uint32_t pc, uint16_t line_number, const uint8_t * p_file_name);`

is now replaced by:

`typedef void (*nrf_fault_handler_t)(uint32_t id, uint32_t pc, uint32_t info);`

The application code must now provide a fault handler in the above format. The source of the fault is provided in the fault ID parameter (**`id`**) and the value of the program counter at the time of the fault is provided in the program counter parameter (**`pc`**) . So far the SoftDevice defines the following fault IDs:

- **`NRF_FAULT_ID_SD_ASSERT`**: The SoftDevice has triggered an assertion. Record the value of the `pc` parameter and make it available to the Nordic support team to start an internal investigation.
- **(s132 only)** **`NRF_FAULT_ID_APP_MEMACC`**: The application has triggered an unallowed memory access. The value of the `pc` param eter will contain the address of the instruction that executed the invalid memory access, or the address of the instruction following the violation. To find out the filename and line number within your application source code that correspond to the `pc` you can use the appropriate tool provided with your toolchain. For example if your linker outputs files in the ELF format you can use the addr2line tool which is part of the GNU ARM Embedded toolchain for this purpose. Note that you don't need to have compiled with GCC to use addr2line, and that there are several common filename extensions for ELF files, e.g. .elf, and .axf.

```
// Syntax
arm-none-eabi-addr2line <pc> -e application.elf

// Example, pc=0x01da6a
$ arm-none-eabi-addr2line 0x01da6a -e app_beacon.elf
C:\dev\app_beacon\src\main.c:34
```

Please note that as part of this transition from asserts to faults the previously distributed **`softdevice_assert.h`** file is no longer part of the public API.

Oscillator configuration

The configuration of the 32 kHz RCOSC calibration in `sd_softdevice_enable()` has been made more flexible. It now supports more calibration intervals, and the ability to combine temperature and time triggered calibration.

`sd_softdevice_enable(nrf_clock_lf_cfg_t const * p_clock_lf_cfg, nrf_fault_handler_t fault_handler));`

```
// Example configuration equivalent to the old
NRF_CLOCK_LFCLKSRC_RC_250_PPM_TEMP_1000MS_CALIBRATION
nrf_clock_lf_cfg_t rc_cfg = {
    .source = NRF_CLOCK_LF_SRC_RC,
    .rc_ctiv = 4,      // Check temperature every 4 * 250ms
    .rc_temp_ctiv = 1, // Only calibrate if temperature has changed.
};

sd_softdevice_enable(&rc_cfg, &app_fault_handler);

// Example configuration equivalent to the old NRF_CLOCK_LFCLKSRC_XTAL_75_PPM
nrf_clock_lf_cfg_t xtal_cfg = {
    .source = NRF_CLOCK_LF_SRC_XTAL,
    .xtal_accuracy = NRF_CLOCK_LF_XTAL_ACCURACY_75_PPM
};

sd_softdevice_enable(&xtal_cfg, &app_fault_handler);

// Recommended configuration for using the RC oscillator with s132 (see nrf_sdm.h
for details)
nrf_clock_lf_cfg_t rc_cfg = {
    .source = NRF_CLOCK_LF_SRC_RC,
    .rc_ctiv = 16,     // Check temperature every 4 seconds
    .rc_temp_ctiv = 2, // Calibrate at least every 8 seconds even if the
temperature hasn't changed
};

sd_softdevice_enable(&rc_cfg, &app_fault_handler);
```

App priorities

The enumeration `NRF_APP_PRIORITIES` has been removed. Application developers must use the interrupt priority levels directly instead.

For s130 the interrupt priority levels available to the application are: **1** and **3**.

For s132 the interrupt priority levels available to the application are: **2**, **3**, **6** and **7**.

SEVONPEND flag and high interrupt priorities

Applications must **not** modify the `SEVONPEND` flag in the `SCR` register when running in priority level 1 for s130 and priority levels 2 or 3 for s132.

Type definitions

Type definitions for certain basic types have been removed. The following type definitions must be replaced with `uint8_t`:

`nrf_power_mode_t`, `nrf_power_failure_threshold_t`, `nrf_radio_notification_distance_t`, `nrf_radio_notification_type_t`

and the following must be replaced with `uint32_t`:

`nrf_app_irq_priority_t` `nrf_power_dcdc_mode_t`

MBR size

The macro `MBR_SIZE` has been moved to `nrf_mbr.h`.

Changes to the sd_nvic_* API

The `sd_nvic_*` API functions have changed from being SV calls into the SoftDevice to being static functions implemented in a new header file, **`nrf_nvic.h`**. This header file must be included in all source files that call any API function than begins with `sd_nvic_`. If a project includes **`nrf_nvic.h`** in any of its source files, one of them must declare and zero initialize a global instance of **`nrf_nvic_state_t`** in the form:

```
nrf_nvic_state_t nrf_nvic_state = {0};
```

Flash protection

The flash protection API now takes 4 parameters, only the first 2 of which are applicable for the s130:

```
sd_flash_protect(uint32_t block_cfg0, uint32_t block_cfg1, uint32_t block_cfg2, uint32_t block_cfg3);
```

Radio Timeslot API macro changes

The macros for high frequency clock configuration have been renamed in the Radio Timeslot API:

- `NRF_RADIO_HFCLK_CFG_DEFAULT` and `NRF_RADIO_HFCLK_CFG_FORCE_XTAL`
- are now **`NRF_RADIO_HFCLK_CFG_XTAL_GUARANTEED`** and **`NRF_RADIO_HFCLK_CFG_NO_GUARANTEE`**

The default is now **`NRF_RADIO_HFCLK_CFG_XTAL_GUARANTEED`** which guarantees that the high frequency clock source is the crystal for the whole duration of the timeslot. This should be the preferred option for events that use the radio or require high timing accuracy.

SoftDevice runtime configuration

The number of Vendor Specific UUIDs, connection count and bandwidth are now configurable on `sd_ble_enable()` using the new parameters in the substructures of `ble_enable_params_t.` Those new parameters are listed below:

- **`vs_uuid_count`**: The number of Vendor Specific UUID bases that the SoftDevice will reserve space for. Formerly this number was fixed and set to `BLE_UUID_VS_MAX_COUNT`.
- **`p_conn_bw_counts`**: The optional connection bandwidth configuration structure. This determines the amount of memory that the SoftDevice will reserve for packets. See the bandwidth configuration section for more details.
- **`periph_conn_count`**: The total amount of concurrent connections as a peripheral that will be available to the application.
- **`central_conn_count`**: The total amount of concurrent connections as a central that will be available to the application.
- **`central_sec_count`**: The total amount of concurrent pairing procedures that will be available to the application to be shared among all connections as a central.

If the maximum number of connections supported by the SoftDevice is exceeded in the call to `sd_ble_enable()` the SoftDevice will return **NRF_ERROR_CONN_COUNT**.

SoftDevice RAM usage

At runtime the IC's RAM is split into 2 regions: The SoftDevice RAM region (between `0x20000000` and APP_RAM_BASE-1) and the application RAM region (between APP_RAM_BASE and the call stack). The start address of the application RAM region (APP_RAM_BASE) is dependent on the configuration provided to the SoftDevice in the call to `sd_ble_enable()`.

The `sd_ble_enable()` call has a new parameter.

- `uint32_t sd_ble_enable(ble_enable_params_t * p_ble_enable_params)`
- **`uint32_t sd_ble_enable(ble_enable_params_t * p_ble_enable_params, uint32_t * p_app_ram_base)`**

The new **`*p_app_ram_base`** parameter should be set by the application to APP_RAM_BASE. The SoftDevice will return the minimum APP_RAM_BASE required by the SoftDevice for the configuration. If the APP_RAM_BASE provided by the application is smaller than the APP_RAM_BASE returned by the SoftDevice, `sd_ble_enable()` will return `NRF_ERROR_NO_MEM`.

Note: The nRF5 SDK provides definitions for common configurations and several toolchains. You can skip the rest of this section if you plan to use the nRF5 SDK examples directly and do not intend to create new configurations.

The application must **always** provide the current starting address of its RAM area (as defined in the project file, scatter file or linker script) as the **`*p_app_ram_base`** parameter to `sd_ble_enable()`. Failure to do so might result in the SoftDevice overwriting the application memory area and/or memory access violations. Most toolchains provide a linker symbol for the starting address of their RAM area, referred to as __L

INKER_APP_RAM_BASE in this documentation.

The following table shows examples of linker symbols that can define __LINKER_APP_RAM_BASE for different toolchains. The actual value will depend on the project file, scatter file or linker script settings.

Toolchain	__LINKER_APP_RAM_BASE
ARMCC/Keil	Image$$RW_IRAM1$$Base
IAR	__ICFEDIT_region_RAM_start__
GCC	__data_start__

The recommended approach to obtaining and maintaining the required APP_RAM_BASE for the application is the following:

1. In your project file, scatter file or linker script, set the starting address of your application's RAM (APP_RAM_BASE) to at least the minimum APP_RAM_BASE specified in the release notes.
2. In your application's source code, set the value of **`*p_app_ram_base`** to `__LINKER_APP_RAM_BASE`.
3. Set the desired parameters to be provided to `sd_ble_enable()`.
4. Compile, link and run the application.
5. If the amount of memory assigned to the SoftDevice by **`*p_app_ram_base`** is large enough to fit the configuration, `sd_ble_enable()` will return `NRF_SUCCESS`, otherwise it will return `NRF_ERROR_NO_MEM`.
6. On return of `sd_ble_enable()`, **`*p_app_ram_base`** will contain the APP_RAM_BASE required for the given configuration.
7. In your project file, scatter file or linker script, update the starting address of your application's RAM (APP_RAM_BASE) to **`*p_app_ram_base`** from step 6, and recompile the application.

Please note that it is possible to run the application with `APP_RAM_BASE` set higher than the minimum required by the selected configuration. Doing so will result in an area of memory being unused located between the SoftDevice's and the application's memory areas.

Enabling the BLE Stack

```
ble_enable_params_t params;
uint32_t retv;
uint32_t app_ram_base;

memset(&params, 0x00, sizeof(params));
/* set the number of Vendor Specific UUIDs to 5 */
params.common_enable_params.vs_uuid_count = 5;
/* this application requires 1 connection as a peripheral */
params.gap_enable_params.periph_conn_count = 1;
/* this application requires 3 connections as a central */
params.gap_enable_params.central_conn_count = 3;
/* this application only needs to be able to pair in one central link at a time */
params.gap_enable_params.central_sec_count = 1;
/* we require the Service Changed characteristic */
params.gatts_enable_params.service_changed = 1;
/* the default Attribute Table size is appropriate for our application */
params.gatts_enable_params.attr_tab_size = BLE_GATTS_ATTR_TAB_SIZE_DEFAULT;

/* set app_ram_base to the starting memory address of the application RAM,
   obtained directly from the linker */
app_ram_base = __LINKER_APP_RAM_BASE;
/* enable the BLE Stack */
retv = sd_ble_enable(&params, &app_ram_base);
if(retv == NRF_SUCCESS)
{
 /* Verify that __LINKER_APP_RAM_BASE matches the SD calculations */
 if(app_ram_base != __LINKER_APP_RAM_BASE)
 {
  /* The application's starting RAM address is higher than the one required for
this configuration.
     An area of memory will remain unused between the SoftDevice and the
application memory areas.
           To detect this, place a breakpoint here and/or output (app_ram_base)
           through a debug interface.
   */
 }
}
else if(retv == NRF_ERROR_NO_MEM)
{
 /* The application's starting RAM address is lower than the one required for this
configuration.
       This is an unrecoverable error because the SoftDevice and the application
memory areas overlap.
       To detect this, place a breakpoint here and/or output (app_ram_base)
       through a debug interface.
 */
 while(1){}
}
```

Default Attribute Table size changed

The default Attribute Table size has gone down from `0x600` bytes to `0x580` bytes. If the application is not setting a custom Attribute Table size and it is filling it completely, it will now need to configure a larger, non-default memory area size dedicated to it (`ble_gatts_enable_params_t::attr_tab_size`) in the call to sd_ble_enable().

(s130 only) CPU and Radio mutual exclusion option removed

The `BLE_COMMON_OPT_RADIO_CPU_MUTEX` option is no longer part of the SoftDevice API so applications making use of it will need to remove all code using it. The option is no longer necessary since this version of the SoftDevice is only compatible with IC revision 3 of the nRF51 series, which no longer requires mutual exclusion between the radio and the CPU during operation.

TX packet management

The user TX packet management has been modified to adapt it to the fact that different connections can now make different packet counts available to the application, depending on the role and bandwidth configuration. This means that the application now needs to obtain the TX packet count **after** each connection is established, and needs also to keep an independent variable for the TX packet count of each connection.

The prototype has been therefore renamed and adapted:

- `uint32_t sd_ble_tx_buffer_count_get(uint8_t *p_count)`
- **`uint32_t sd_ble_tx_packet_count_get(uint16_t conn_handle, uint8_t *p_count)`**

Here's an example of an application obtaining the TX packet count for a particular connection and storing it in a global variable for later use:

```
case BLE_GAP_EVT_CONNECTED:
 uint8_t count;
 uint16_t conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
 sd_ble_tx_packet_count_get(conn_handle, &count);
 /* store TX packet count for later use */
 tx_packet_counts[conn_handle] = count;
 break;
```

TX power setting

The `sd_ble_gap_tx_power_set()` SV call now accepts the following values as the lowest power setting:

- s130: -30dBm
- s132: -40dBm

If the application code made use of values different from those in its minimum power output mode it will have to be adapted it to conform with the changes.

Additional link field in the key distribution bitfield

The `ble_gap_sec_kdist_t` bitfield now includes an additional **`link`** bit. This **must always be set to 0** by the application since it is only intended for use with dual-mode BR/EDR+BLE solutions.

Additional lesc field in the encryption information structure

A new **`lesc`** bit has been added to the `ble_gap_enc_info_t` structure. It is important to initialize this bit correctly when loading stored security keys, so that the SoftDevice can set the connection's security level accordingly.

Additional fields in the security parameters

Two new fields have been added to `ble_gap_sec_params_t`:

- **`lesc`**: This enables LE Secure Connections locally when starting a pairing or bonding procedure. If the application does not wish to use LE Secure Connections and instead use legacy pairing simply set this bit to 0.
- **`keypress`**: This enables keypress notifications locally when starting a pairing or bonding procedure. Keypress notifications can be used whenever the Passkey Entry pairing method is selected, both in legacy pairing or LE Secure Connections.

Both fields need to be initialized to the desired value by applications transitioning to this SoftDevice version.

Security keys identification by locality instead of by GAP role

The security keys included in `ble_gap_sec_keyset_t` are no longer identified by GAP role, but rather by associating them with the local device (own) or with the remote device (peer):

- `ble_gap_sec_keyset_t::keys_periph` and `ble_gap_sec_keyset_t::keys_central` are now expressed in terms of `ble_gap_sec_keyset_t::`**`keys_own`** and `ble_gap_sec_keyset_t::`**`keys_peer`**
- `ble_gap_sec_params_t::kdist_periph` and `ble_gap_sec_params_t::kdist_central` are now expressed in terms of `ble_gap_sec_params_t::`**`kdist_own`** and `ble_gap_sec_params_t::`**`kdist_peer`**
- `ble_gap_evt_auth_status_t::kdist_periph` and `ble_gap_evt_auth_status_t::kdist_central` are now expressed in terms of `ble_gap_evt_auth_status_t::`**`kdist_own`** and `ble_gap_evt_auth_status_t::`**`kdist_peer`**

For example, a multi-role application wanting to distribute its own LTK when acting as a peripheral, but only its IRK when acting as a central and that always accepts IRKs from the peer no matter the role:

```
/* Connected */
if(own_role == BLE_GAP_ROLE_PERIPH)
{
 sec_params.kdist_own.enc = 1;
}
else /* BLE_GAP_ROLE_CENTRAL */
{
 sec_params.kdist_own.id = 1;
}
sec_params.kdist_peer.id = 1;
```

Identity key distribution change

When distributing Identity keys during a bonding procedure, the handling of the pointers within the `ble_gap_sec_keyset_t` structure has changed in the following manner:

- Setting `ble_gap_sec_keyset_t::keys_own::p_id_key` to `NULL` remains unchanged: the stack will continue to make use of the currently set Bluetooth address and IRK and distribute them to the peer, but the application will not receive a copy in its memory
- Setting `ble_gap_sec_keyset_t::keys_own::p_id_key` to a valid pointer to a location in the application memory will distribute the same Bluetooth address and IRK as above (the currently set ones) and also make them available to the application

That means that if the application distributed a custom Bluetooth address and IRK using the following deprecated functionality:

```
/* Connected */
keyset.keys_own.p_id_key = &app_custom_id_key;
keyset.keys_own.p_id_addr_info = &custom_bdaddr;
sd_ble_gap_sec_params_reply(conn_handle, BLE_GAP_SEC_STATUS_SUCCESS, &sec_params,
&keyset);
```

it will now have to manually set those before calling `sd_ble_gap_sec_params_reply()`:

```
/* Connected */
ble_opt_t opt;
sd_ble_gap_address_set(BLE_GAP_ADDR_CYCLE_MODE_NONE,
&app_custom_id_key.id_addr_info);
opt.gap_opt.privacy.p_irk = &app_custom_id_key.id_info;
opt.gap_opt.privacy.interval_s = APP_ADDR_REFRESH_S;
sd_ble_opt_set(BLE_GAP_OPT_PRIVACY, &opt);
keyset.keys_own.p_id_key = &distributed_id_key;
sd_ble_gap_sec_params_reply(conn_handle, BLE_GAP_SEC_STATUS_SUCCESS, &sec_params,
&keyset);
```

GATT Server Read/Write events: attribute context removed

The `ble_gatts_attr_context_t` type has been removed from the GATT Server API. The two structures that included an instance of it as a member now include instead a `ble_uuid_t` instance to identify the attribute:

- `ble_gatts_evt_write_t::context` has been replaced by `ble_gatts_evt_write_t::`**`uuid`**
- `ble_gatts_evt_read_t::context` has been replaced by `ble_gatts_evt_read_t::`**`uuid`**

In practical usage most applications store the handles associated with a particular characteristic when populating the Attribute Table. Calculating the context for each incoming read or write operation was an expensive and time-consuming task, and therefore the field has been removed and instead replaced by the attribute UUID. The combination of attribute handle and attribute UUID provided in the event structure should be enough for the application to identify the attribute within the set that has been previously populated.

GATT Server Authorizable Write Commands

Whenever the application enables write authorization for a characteristic value or a descriptor in the Attribute Table (`ble_gatts_attr_md_t::wr_auth`), all incoming write operations will now require application authorization. In particular this now includes Write Commands (also called Write Without Response) which will arrive in the same event form (`BLE_GATTS_EVT_WRITE`) but with a new field set (`ble_gatts_evt_write_t::`**`auth_required`**) to indicate to the application that the data has not been written into the Attribute Table. Upon handling of the event the application can decide whether it wants to write the incoming data to the Attribute Table using `sd_ble_gatts_value_set()` or discard it.

Handling incoming authorizable Write Commands

```
case BLE_GATTS_EVT_WRITE:
 uint16_t conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;
 uint16_t attr_handle = p_ble_evt->evt.gatts_evt.params.write.handle;
 uint16_t offset = p_ble_evt->evt.gatts_evt.params.write.offset;
 uint8_t *p_data = p_ble_evt->evt.gatts_evt.params.write.data;
 uint16_t dlen = p_ble_evt->evt.gatts_evt.params.write.len;
 if(p_ble_evt->evt.gatts_evt.params.write.auth_required)
 {
  /* incoming write command on an attribute requiring authorization,
           validate the incoming data pointed to by p_data */
  if(app_data_authorize(p_data, offset, dlen))
  {
            /* the application manually writes the incoming data (p_data) to the
Attribute Table */
   ble_gatts_value_t value;
   value.len = dlen;
   value.offset = offset;
   value.p_value = p_data;
   sd_ble_gatts_value_set(conn_handle, attr_handle, &value);
  }
 }
 break;
```

GATT Server Write Authorization and peer data

Applications making use of authorization to handle incoming write operations, and in particular Write Requests and app-handled Queued Writes, will now have to store the incoming data to be provided later to the SoftDevice. Depending on how the application handles the authorization procedure, this can be done by providing the same pointer contained in the event field, or copying the data into a temporary storage area if required.

- Authorizing directly in the event handler:

```
case BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST:
 if(p_ble_evt->evt.gatts_evt.params.authorize_request.type ==
BLE_GATTS_AUTHORIZE_TYPE_WRITE)
 {
  uint16_t conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;
  uint16_t offset =
p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.offset;
  uint16_t dlen =
p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.len;
  uint8_t *p_data =
p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.data;
  /* incoming write command on an attribute requiring authorization, validate the
data */
  if(app_data_authorize(p_data, offset, dlen))
  {
   ble_gatts_rw_authorize_reply_params_t auth_reply;
   auth_reply.type = BLE_GATTS_AUTHORIZE_TYPE_WRITE;
   auth_reply.params.write.gatt_status = BLE_GATT_STATUS_SUCCESS;
   auth_reply.params.write.update = 1;
   auth_reply.params.write.offset = offset;
   auth_reply.params.write.len = dlen;
            /* reuse the same pointer obtained from the event */
   auth_reply.params.write.p_data = p_data;

   sd_ble_gatts_rw_authorize_reply(conn_handle, &auth_reply);
  }
 }
 break;
```

- Authorizing outside of the event handler:

```
/* global variable storing the authorization data */
struct
{
 uint16_t conn_handle;
 uint16_t offset;
 uint16_t dlen;
 uint8_t data[MAX_DATA];
} auth_write;

[..]

case BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST:
 if(p_ble_evt->evt.gatts_evt.params.authorize_request.type ==
BLE_GATTS_AUTHORIZE_TYPE_WRITE)
 {
  /* store the metadata */
  auth_write.conn_handle = p_ble_evt->evt.gatts_evt.conn_handle;
  auth_write.offset =
p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.offset;
  auth_write.dlen =
p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.len;
  /* store the actual incoming data */
  memcpy(&auth_write.data,
&p_ble_evt->evt.gatts_evt.params.authorize_request.request.write.data,
auth_write.dlen);
 }
 break;

[..]

/* authorization complete */
ble_gatts_rw_authorize_reply_params_t auth_reply;
auth_reply.type = BLE_GATTS_AUTHORIZE_TYPE_WRITE;
auth_reply.params.write.gatt_status = BLE_GATT_STATUS_SUCCESS;
auth_reply.params.write.update = 1;
/* obtain the data */
auth_reply.params.write.offset = auth_write.offset;
auth_reply.params.write.len = auth_write.dlen;
auth_reply.params.write.p_data = auth_write.data;

sd_ble_gatts_rw_authorize_reply(auth_write.conn_handle, &auth_reply);
```

New functionality

Configurable bandwidth

The connections can now be configured to have low, medium or high bandwidth. This can be specified for both TX and RX independently to allow for asymmetric bandwidth. This is an optional feature and if the application chooses not to use it the SoftDevice will instead configure the connections with defaults. The default configuration for connections as a central is `BLE_CONN_BW_MID` (both for TX and RX), and for connections as a peripheral it is `BLE_CONN_BW_HIGH` (both for TX and RX).

When using the configurable bandwidth option the application should have specified beforehand, at BLE stack initialization time, a set of connection bandwidth configurations that includes the ones that it intends to use through this option. Once a bandwidth configuration for a particular role is chosen through the `sd_ble_opt_set()` SV call, all connections of that role established from that time on will use the chosen configuration until a new one is set.

Additional information about this topic can be found in the SoftDevice Specification at http://infocenter.nordicsemi.com/.

The following table shows an approximate comparison of connections and bandwidth configuration for previous SoftDevices as well as the the s13x v2.0.1 configured as shown in the example below.

	connections as a peripheral		connections as a central	
	number	RX / TX bandwith	number	RX / TX bandwith
s110 v8.0	1	HIGH / HIGH	0	-
s120 v2.1 (peripheral mode)	1	HIGH / HIGH	0	-
s120 v2.1 (central mode)	0	-	8	LOW / LOW
s130 v1.0	1	MID / MID	3	LOW / LOW
s13x v2.0.1 (default)	0	HIGH / HIGH	0	MID / MID
s13x v2.0.1 (example configuration below)	1	MID / MID	1	HIGH / MID

```
/* Example for one medium-bandwidth RX and TX connection as a peripheral and
high-bandwidth RX, medium-bandwidth TX connection as a central. */
ble_conn_bw_counts_t conn_bw_counts = {
  .tx_counts = {.high_count = 0, .mid_count = 2, .low_count = 0},
  .rx_counts = {.high_count = 1, .mid_count = 1, .low_count = 0}
};

ble_enable_params_t enable_params = {0};
enable_params.common_enable_params.p_conn_bw_counts = &conn_bw_counts;
enable_params.gap_enable_params.central_conn_count = 1;
enable_params.gap_enable_params.periph_conn_count = 1;

sd_ble_enable(&enable_params, ... );


ble_opt_t ble_opt;

/* Configure bandwidth and connect as a peripheral */
ble_common_opt_conn_bw_t conn_bw = { .role = BLE_GAP_ROLE_PERIPH, .conn_bw = {
.conn_bw_rx = BLE_CONN_BW_MID, .conn_bw_tx = BLE_CONN_BW_MID } };
ble_opt.common_opt.conn_bw = conn_bw;
sd_ble_opt_set(BLE_COMMON_OPT_CONN_BW, &ble_opt);
sd_ble_gap_adv_start( ... );

/* Connection established with the configured bandwidth */

/* Configure bandwidth and connect as a central */
ble_common_opt_conn_bw_t conn_bw = { .role = BLE_GAP_ROLE_CENTRAL, .conn_bw = {
.conn_bw_rx = BLE_CONN_BW_HIGH, .conn_bw_tx = BLE_CONN_BW_MID } };
ble_opt.common_opt.conn_bw = conn_bw;
sd_ble_opt_set(BLE_COMMON_OPT_CONN_BW, &ble_opt);
sd_ble_gap_connect( ... );


/* Connection established with the configured bandwidth */
```

Block encryption

The blocking block encryption SV call `sd_ecb_block_encrypt()` depends on the hardware encryption block and therefore will require to wait for it to complete before it returns to the application. If the user now sets the SEVONPEND bit in the SCR to 1 before calling this function, the SoftDevice will sleep while the ECB is running instead of entering a busy loop.

A second SV call has also been introduced to perform multiple block encrypt operations in a single SV call to avoid the context switch overhead when more than one block of data needs to be encrypted.

`uint32_t sd_ecb_blocks_encrypt(uint8_t block_count, nrf_ecb_hal_data_block_t * p_data_blocks);`

sd_ecb_blocks_encrypt() example usage

```
/* global variable storing the authorization data */

nrf_ecb_hal_data_block_t blocks[ECB_BLOCK_COUNT];

/* intialize data blocks */
for(i = 0; i < ECB_BLOCK_COUNT; i++)
{
 blocks[i].p_key = &app_keys[i];
 blocks[i].p_cleartext = &app_cleartext[i];
 blocks[i].p_ciphertext = &app_dest[i];
}

sd_ecb_blocks_encrypt(ECB_BLOCK_COUNT, blocks);
```

PA/LNA support

A new BLE option, **`BLE_COMMON_OPT_PA_LNA`**, and its corresponding option structure, **`ble_common_opt_pa_lna_t`**, have been added to provide support for power amplifiers and low noise amplifiers. The application is responsible for correctly initializing the option parameter structure with the required fields that map to the hardware design:

- PA and LNA pins and active level
- Set and Clear PPI channel IDs
- GPIOTE channel ID

PA/LNA option usage

```
/* PA/LNA configuration */
ble_opt_t pa_lna_opt = {
  .common_opt = {
    .pa_lna = {
      .pa_cfg = {
        .enable       = 1,
        .active_high  = 1,
        .gpio_pin     = APP_PA_PIN /* GPIO connected to the PA control pin */
      },
      .lna_cfg = {
        .enable       = 1,
        .active_high  = 1,
        .gpio_pin     = APP_LNA_PIN /* GPIO connected to the LNA control pin */
      },
      .ppi_ch_id_set  = APP_AMP_PPI_CH_ID_SET, /* PPI channel the app gives the SD
to set the pins */
      .ppi_ch_id_clr  = APP_AMP_PPI_CH_ID_CLR, /* PPI channel the app gives the SD
to clear the pins */
      .gpiote_ch_id   = APP_AMP_GPIOTE_CH_ID   /* GPIOTE channel the app gives the
SD to control the pins */
    }
  }
};

sd_ble_opt_set(BLE_COMMON_OPT_PA_LNA, &pa_lna_opt);
```

LE Secure Connections

Version 4.2 of the Bluetooth Specification introduced a new mode of operation for the Security Manager Protocol, which enables the use of Public Key Cryptography for the generation of security keys. This means that applications can now select the mode of operation of the Security Manager when performing a pairing or bonding procedure:

- Legacy pairing: Set the **`lesc`** bit in `ble_gap_sec_params_t` to **0**.
- LE Secure Connections: Set the **`lesc`** bit in `ble_gap_sec_params_t` to **1**.

Please note that, in order for LE Secure Connections to be used, the peer will need to support it. If not, legacy pairing will be used by default.

The SoftDevice implements the Security Manager Protocol and cryptographic toolbox functionality required to enable LE Secure Connections, but it does **not** include the Elliptic Curve Cryptography (ECC) methods required to generate public keys and shared secrets. This means that applications must include their own implementation of ECC. The SoftDevice never requires knowledge of the application's private key, since it delegates the calculation of the shared secret (DHKey) to the application itself:

- `ble_gap_sec_keys_t::`**`p_pk`** (*own* only) is provided by the application and represents the P-256 public key (PK_{own}) that matches the local private key (SK_{own}). The key is provided as a part of the `ble_gap_sec_keyset_t` structure when calling `sd_ble_gap_sec_params_reply()`.
- **`BLE_GAP_EVT_LESC_DHKEY_REQUEST`** is a new event requesting the application to calculate the shared secret, which is the result of P256(SK_{own}, PK_{peer}). The event structure contains the peer's public key (PK_{peer}) so that the application can start the calculation of the DHKey. Once the application has completed the calculation it must communicate the result to the SoftDevice by using the new **`sd_ble_gap_lesc_dhkey_reply()`** SV call.

Additional API changes introduced by LE Secure Connections:

- `ble_gap_evt_passkey_display_t` now contains an additional field, **`match_request`**, used for the new Numeric Comparison pairing algorithm
- `sd_ble_gap_auth_key_reply()` now accepts BLE_GAP_AUTH_KEY_TYPE_PASSKEY coupled with a `NULL` `p_key` pointer to indicate a match in the new Numeric Comparison pairing algorithm
- **`sd_ble_gap_lesc_oob_data_get()`** and **`sd_ble_gap_lesc_oob_data_set()`** have been introduced to support the new LE

Secure Connections OOB pairing method, which is substantially different from the Legacy OOB version

Additional details can be found in the API documentation and the Message Sequence Charts (MSCs) included with the SoftDevice.

Passkey entry keypress notifications

During pairing procedures using the Passkey Entry pairing algorithm (both in Legacy and LE Secure Connections modes) it is now possible to provide feedback to the peer regarding the keypresses being input by the user. The actual value of the keypresses is never sent over the air, but the notifications can be sent to provide visual feedback of the keys being pressed.

To enable the application to send keypress notifications to the peer, the following SV call has been introduced:

- **`sd_ble_gap_keypress_notify(uint16_t conn_handle, uint8_t kp_not)`**

Where kp_not maps to any of the values present in the **`BLE_GAP_KP_NOT_TYPES`** enumeration.

Sending keypress notifications

```
/* Pairing procedure using the Passkey Entry algorithm in progress, local device
inputs passkey */

/* User starts entering the passkey */
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_START);
/* User inputs digits */
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_IN);
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_IN);
/* User deletes a digit */
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_OUT);
/* User clears the input completely */
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_CLEAR);
/* User ends the input procedure */
sd_ble_gap_keypress_notify(conn_handle, BLE_GAP_KP_NOT_TYPE_PASSKEY_END);
```

Please note that **`sd_ble_gap_keypress_notify()`** can return `NRF_ERROR_BUSY` if the application calls it too often and the previous keypress notification has not made it over the air to the peer yet. In that case the application should queue the keypresses internally and retry at a later time.

A new event has also been added to allow the application to receive keypress notifications from the peer:

- **`BLE_GAP_EVT_KEY_PRESSED`** and its corresponding **`ble_gap_evt_key_pressed_t`**

Receiving keypress notifications

```
/* Pairing procedure using the Passkey Entry algorithm in progress, peer device
inputs passkey */

/* handle the event */
case BLE_GAP_EVT_KEY_PRESSED:
 switch(p_ble_evt->evt.gap_evt.params.key_pressed.kp_not)
 {
 case BLE_GAP_KP_NOT_TYPE_PASSKEY_START:
  /* Remote user has started entering the passkey */
  break;
 case BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_IN:
  /* Remote user has input a digits */
  break;
 case BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_OUT:
  /* Remote user has deleted a digit */
  break;
 case BLE_GAP_KP_NOT_TYPE_PASSKEY_CLEAR:
  /* Remote user has cleared the input completely */
  break;
 case BLE_GAP_KP_NOT_TYPE_PASSKEY_END:
  /* Remote user has ended the input procedure */
  break;
 }
```

Security Mode 1 Level 4

A new security level has been introduced along with support for LE Secure Connections. Security levels are used in GAP and GATT Server to define the connection's security level and the access requirements for the peer to read and write attributes in the local Attribute Table. The list of supported security levels is now:

- Security Mode 0, Level 0: No access allowed regardless of the connection's security level
- Security Mode 1, Level 1: No encryption. The peer can read and write the attribute without restrictions
- Security Mode 1, Level 2: Encryption without MITM protection. Access to the attribute requires an encrypted connection (Legacy or LE Secure Connections) with or without MITM protection
- Security Mode 1, Level 3: Encryption with MITM protection. Access to the attribute requires an encrypted connection (Legacy or LE Secure Connections) with MITM protection
- **Security Mode 1, Level 4: LESC Encryption with MITM protection. Access to the attribute requires an encrypted connection (LE Secure Connections only) with MITM protection**

To honour the new security level (Security Mode 1, Level 4) encryption must be enabled with an LTK that has been generated during a pairing or bonding procedure using LE Secure Connections with MITM protection (Numeric Comparison, Passkey Entry or OOB). This is the highest security level available when defining the access requirements (permissions) of attributes in the local Attribute Table.

A new macro has been made available to set `ble_gap_conn_sec_mode_t` to the new security level:

`BLE_GAP_CONN_SEC_MODE_SET_LESC_ENC_WITH_MITM`

An additional Advertising Data type has been added to `ble_gap.h`

`BLE_GAP_AD_TYPE_URI`

GATT Client attribute info discovery

A new SV call allows applications to obtain basic attribute information from the peer's Attribute Table:

```
uint32_t sd_ble_gattc_attr_info_discover(uint16_t conn_handle, ble_gattc_handle_range_t const *
p_handle_range);
```

the matching event identifier and structure are also part of this new feature:

- **`BLE_GATTC_EVT_ATTR_INFO_DISC_RSP`**
- **`ble_gattc_attr_info_t`**
- **`ble_gattc_evt_attr_info_disc_rsp_t`**

This is the only GATT Client function that allows the application to retrieve full 128-bit UUIDs that do **not** need to be part of the list populated with `sd_ble_vs_uuid_add()`. An example of 128-bit UUID retrieval is shown below.

128-bit UUID retrieval using sd_ble_gatt_attr_info_discover()

```
ble_gattc_handle_range_t handle_range;


/* list all attributes on the peer's Attribute Table */
handle_range.start_handle = 0x0001;
handle_range.end_handle = 0xFFFF;
sd_ble_gattc_attr_info_discover(conn_handle, &handle_range);

[..]

/* handle the event */
case BLE_GATTC_EVT_ATTR_INFO_DISC_RSP:
 /* check if we have 128-bit UUIDs */
 if(p_ble_evt->evt.gattc_evt.params.attr_info_disc_rsp.format ==
BLE_GATTC_ATTR_INFO_FORMAT_128BIT)
 {
  uint16_t attr_handle;
  ble_uuid128_t uuid128;
  /* Obtain the attribute handle and the full 128-bit UUID */
  attr_handle=
p_ble_evt->evt.gattc_evt.params.attr_info_disc_rsp.attr_info[0].handle;
  memcpy(&uuid128,
&p_ble_evt->evt.gattc_evt.params.attr_info_disc_rsp.attr_info[0].info.uuid128.uuid1
28, sizeof(uuid128));
 }
 break;
```

GATT Server first user attribute handle retrieval

When using the Service Changed characteristic to indicate to the peer that the local Attribute Table structure has changed, it is often useful to find out at which handle the application-controlled region of the Attribute Table begins. For that specific purpose a new SV call has been introduced:

`uint32_t sd_ble_gatts_initial_user_handle_get(uint16_t *p_handle);`

This allows the application to communicate to the peer the exact range of the attributes that require rediscovery.

Obtaining the first user handle to indicate a Service Changed

```
uint16_t first_attr_handle;

sd_ble_gatts_initial_user_handle_get(&first_attr_handle);
sd_ble_gatts_service_changed(conn_handle, first_attr_handle, last_affected_handle);
```

GATT Server local attribute metadata retrieval

The GATT Server module has always allowed applications to retrieve the value of any attribute present in the local Attribute Table by means of the `sd_ble_gatts_value_get()` SV call. Now in addition applications can also retrieve the UUID and metadata of any local attribute using the new function:

`uint32_t sd_ble_gatts_attr_get(uint16_t handle, ble_uuid_t * p_uuid, ble_gatts_attr_md_t * p_md);`

This can be useful in several scenarios, one of which is calculating or verifying the structure of the local Attribute Table regardless of the current attribute values, focusing instead only in the layout itself

Obtaining the UUID and metadata of all local attributes

```
uint16_t attr_handle;
ble_uuid_t uuid;
ble_gatts_attr_md_t attr_md;


/* start at the first valid user attribute handle */
sd_ble_gatts_initial_user_handle_get(&attr_handle);

/* traverse the Attribute Table obtaining the UUID and metadata for each attribute
*/
while(sd_ble_gatts_attr_get(attr_handle, &uuid, &attr_md) == NRF_SUCCESS)
{
 /* use the uuid and attr_md here */
 attr_handle++;
}
```

GATT Server user memory layout for system attributes

The data format used by the GATT Server to store system attribute data is now fully documented in the API documentation for applications that need to parse it. The data format is used by the following 2 functions:

- `sd_ble_gatts_sys_attr_set()`
- `sd_ble_gatts_sys_attr_get()`

The format documentation applies to the data pointed to by the `p_sys_attr_data` pointer in both of the functions above.

```
/* Renamed GAP SVCs */
```

Long ATT MTU related API changes:

- A new BLE initialization structure, ble_gatt_enable_params_t, for setting the maximum ATT_MTU size the SoftDevice can send or receive.

- A new SV call, sd_ble_gattc_exchange_mtu_request, for starting an ATT_MTU exchange.

- A new event, BLE_GATT_EVT_MTU_EXCHANGED, indicating the completion of an ATT_MTU exchange and the applied ATT_MTU size.

- SV calls sd_ble_gatts_hvx and sd_ble_gatts_service_changed return NRF_ERROR_INVALID_STATE if an ATT_MTU exchange is ongoing.

Here is an example code for using the Long ATT_MTU feature.

```
ble_enable_params_t enable_params = {0};

/* Set maximum ATT_MTU size the SoftDevice can send or receive */
enable_params.gatt_enable_params.att_mtu = 158;

/* Set other BLE Initialization parameters */
...

/* Enable the BLE Stack */
sd_ble_enable(&enable_params, ... );


[..]


uint16_t conn_handle;

/* Establish connection */
...


[..]


/* Start ATT_MTU exchange */
sd_ble_gattc_exchange_mtu_request(conn_handle);


[..]


uint16_t att_mtu;
```

```
/* Handle the event */
case BLE_GATT_EVT_MTU_EXCHANGED:
  /* Store ATT_MTU for later use */
  att_mtu = p_ble_evt->evt.gatt_evt.params.mtu_exchanged.att_mtu;

/* New ATT_MTU size is now applied to GATT procedures for this connection */
```