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 the changes that need to be done in the application when migrating from an older version of the SoftDevice.
- "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 version 5.1.0 from version 5.0.0, then follow the instructions to migrate to version 5.2.0 from version 5.1.0.

s132_nrf52_7.0.1

This section describes how to use the new features of s132_nrf52_7.0.1 when migrating from s132_nrf52_6.1.1. The s132_nrf52_7.0.1 has changed the API compared to s132_nrf52_6.1.1 which requires applications to be recompiled.

Required changes

The application can no longer use the option `BLE_COMMON_OPT_ADV_SCHED_CFG`. The advertiser will always use improved scheduling. This was previously defined as `ADV_SCHED_CFG_IMPROVED`.

The macros `NRF_SOC_APP_PPI_CHANNELS_SD_DISABLED_MSK`, `NRF_SOC_APP_PPI_CHANNELS_SD_ENABLED_MSK`, NRF_SOC_APP_PPI_GROUPS_SD_DISABLED_MSK, and `NRF_SOC_APP_PPI_GROUPS_SD_ENABLED_MSK` are removed. The application can use the macros `NRF_SOC_SD_PPI_CHANNELS_SD_ENABLED_MSK` and `NRF_SOC_SD_PPI_GROUPS_SD_ENABLED_MSK` to deduce the PPI channels and groups available to the application.

New functionality

Connection event trigger

When enabled, this feature will trigger a task at the start of connection events. The application can configure the SoftDevice to trigger a task every N connection events starting from a given connection event counter.

API Updates

- `sd_ble_gap_next_conn_evt_counter_get()`. This API can be used to retrieve the next connection event counter.
- `sd_ble_gap_conn_evt_trigger_start()`, `sd_ble_gap_conn_evt_trigger_stop()`. These APIs can be used to start and stop triggering a task on connection events.

Usage

The code snippet below illustrates how to configure the SoftDevice to toggle the GPIO pin 13, every second connection event, starting at connection event 10. The code snippet stops the connection event trigger when the connection parameters are updated.

```
void on_ble_evt(const ble_evt_t * p_ble_evt)
{
  if (p_ble_evt->header.evt_id == BLE_GAP_EVT_CONNECTED)
  {
    uint16_t conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
```

```
    ble_gap_conn_event_trigger_t trigger_params;
    trigger_params.ppi_ch_id = 0;
    trigger_params.task_endpoint = &NRF_GPIOTE->TASKS_OUT[0];
    trigger_params.conn_evt_counter_start = 10;
    trigger_params.period_in_events = 2;

    sd_ble_gap_conn_evt_trigger_start(conn_handle, &trigger_params);
  }
  else if (p_ble_evt->header.evt_id == BLE_GAP_EVT_CONN_PARAM_UPDATE)
  {
    uint16_t conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
    sd_ble_gap_conn_evt_trigger_stop(conn_handle);
  }
}

int main(void)
{
  /* Configure GPIOTE */
  NRF_GPIO->DIRSET = (1 << 13);
  NRF_GPIOTE->CONFIG[0] = (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
                        | (13 << GPIOTE_CONFIG_PSEL_Pos)
                        | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);

  /* Enable the BLE Stack and connect device */
  sd_ble_enable(...);
  sd_ble_gap_connect(...);

  [...]
}
```

Configurable inclusion of Central Address Resolution (CAR) characteristic and Peripheral Preferred Connection Parameters (PPCP)

API Updates

- `BLE_GAP_CFG_CAR_INCL_CONFIG`. This allows the application to include or exclude the CAR characteristic from the GAP Service.
- `BLE_GAP_CFG_PPCP_INCL_CONFIG`. This allows the application to include or exclude the PPCP characteristic from the GAP Service.

For the above inclusion configuration APIs, the application can use:

- `BLE_GAP_CHAR_INCL_CONFIG_INCLUDE:` The characteristic is included.
- `BLE_GAP_CHAR_INCL_CONFIG_EXCLUDE_WITH_SPACE:` The characteristic is excluded, but the SoftDevice will reserve the attribute handles which are otherwise used for this characteristic.
- `BLE_GAP_CHAR_INCL_CONFIG_EXCLUDE_WITHOUT_SPACE:` The characteristic is excluded.

When CAR is excluded and the SoftDevice is configured to support the central role:

- It is not possible to distribute own IRK.
- It is not possible to enable privacy.

Usage

The code snippet below illustrates how to configure the SoftDevice to exclude both CAR and PPCP from the GAP Service.

```
int main(void)
{
  ble_cfg_t cfg;


  /* Exclude CAR from GAP service, but reserve the ATT Handles that will otherwise be used up by CAR. */
  cfg.gap_cfg.car_include_cfg = BLE_GAP_CHAR_INCL_CONFIG_EXCLUDE_WITH_SPACE;
  sd_ble_cfg_set(BLE_GAP_CFG_CAR_INCL_CONFIG, &cfg, ..);


  /* Exclude PPCP from GAP service, but reserve the ATT Handles that will otherwise be used up by PPCP. */
  cfg.gap_cfg.ppcp_include_cfg= BLE_GAP_CHAR_INCL_CONFIG_EXCLUDE_WITH_SPACE;
  sd_ble_cfg_set(BLE_GAP_CFG_PPCP_INCL_CONFIG, &cfg, ..);

  /* Enable the BLE Stack. */
  sd_ble_enable(...);

  [...]
}
```

s132_nrf52_6.1.0

This section describes how to use the new features of s132_nrf52_6.1.0 when migrating from s132_nrf52_6.0.0. As with all minor releases, the s132_nrf52_6.1.0 is binary compatible with s132_nrf52_6.0.0. Hence existing applications running on s132_nrf52_6.0.0 need not be recompiled unless the new features are needed. Advertising extensions are now fully tested and qualified features.

New functionality

Support for advertising with up to 255 bytes of advertising data

The SoftDevice now supports advertising up to 255 bytes of advertising data. The macro `BLE_GAP_ADV_SET_DATA_SIZE_EXTENDED_MAX_SUPPORTED` is added to indicate this. For connectable extended advertising, the maximum advertising data size is 238 bytes, as indicated by `BLE_GAP_ADV_SET_DATA_SIZE_EXTENDED_CONNECTABLE_MAX_SUPPORTED`.

Usage

Extended Non-Connectable Non-Scannable Advertising with 255 bytes of Advertising data

```
static uint8_t raw_adv_data_data_buffer[BLE_GAP_ADV_SET_DATA_SIZE_EXTENDED_MAX_SUPPORTED];
static ble_gap_adv_data_t adv_data  =
    {
     .adv_data.p_data = raw_adv_data_data_buffer,
     .adv_data.len = sizeof(raw_adv_data_data_buffer)
    };

int main(void)
{
  uint8_t adv_handle = BLE_GAP_ADV_SET_HANDLE_NOT_SET;
  ble_gap_adv_params_t adv_params =
    {
```

```
        .properties=
        {
         .type=BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_NONSCANNABLE_UNDIRECTED
        },
        .interval              = BLE_GAP_ADV_INTERVAL_MAX,
        .duration              = BLE_GAP_ADV_TIMEOUT_LIMITED_MAX,
        .channel_mask          = {0},
        .max_adv_evts          = 0,
        .filter_policy         = BLE_GAP_ADV_FP_ANY,
        .primary_phy           = BLE_GAP_PHY_1MBPS,
        .secondary_phy         = BLE_GAP_PHY_2MBPS,
      };

    /* Enable the BLE Stack */
    sd_ble_enable(...);

    [...]
    sd_ble_gap_adv_set_configure(&adv_handle, &adv_data, &adv_params);

    /* Start advertising */
    sd_ble_gap_adv_start(adv_handle, BLE_CONN_CFG_TAG_DEFAULT);

    [...]
  }
```

Extended Scannable Advertising with 255 bytes of Scan Response data

```
static uint8_t raw_scan_rsp_data_buffer[BLE_GAP_ADV_SET_DATA_SIZE_EXTENDED_MAX_SUPPORTED];
static ble_gap_adv_data_t adv_data  =
    {
```

```
    .scan_rsp_data.p_data = raw_scan_rsp_data_buffer,
    .scan_rsp_data.len = sizeof(raw_scan_rsp_data_buffer)
   };

int main(void)
{
  uint8_t adv_handle = BLE_GAP_ADV_SET_HANDLE_NOT_SET;
  ble_gap_adv_params_t adv_params =
    {
     .properties=
     {
      .type=BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_SCANNABLE_UNDIRECTED
     },
     .interval              = BLE_GAP_ADV_INTERVAL_MAX,
     .duration              = BLE_GAP_ADV_TIMEOUT_LIMITED_MAX,
     .channel_mask          = {0},
     .max_adv_evts          = 0,
     .filter_policy         = BLE_GAP_ADV_FP_ANY,
     .primary_phy           = BLE_GAP_PHY_1MBPS,
     .secondary_phy         = BLE_GAP_PHY_2MBPS,
    };

  /* Enable the BLE Stack */
  sd_ble_enable(...);

  [...]
  sd_ble_gap_adv_set_configure(&adv_handle, &adv_data, &adv_params);

  /* Start advertising */
  sd_ble_gap_adv_start(adv_handle, BLE_CONN_CFG_TAG_DEFAULT);

  [...]
```

```
}
```

Support for receiving up to 255 bytes of advertising data

The SoftDevice now supports receiving up to 255 bytes of advertising data as a scanner. The macro `BLE_GAP_SCAN_BUFFER_EXTENDED_MAX_SUPPORTED` is added to indicate this.

Usage

```
static uint8_t raw_scan_buffer[BLE_GAP_SCAN_BUFFER_EXTENDED_MAX_SUPPORTED];
static ble_data_t scan_buffer =
    {
     .p_data = raw_scan_buffer,
     .len = sizeof(raw_scan_buffer)
    };
static uint16_t scan_window = 0x00A0; /* Corresponds to 100 ms */

int main(void)
{
  ble_gap_scan_params_t scan_params=
    {
     .extended      = 1,        /* Enable extended scanning to be able to receive large advertising data. */
     .scan_phys     = BLE_GAP_PHY_1MBPS,
     .timeout       = BLE_GAP_SCAN_TIMEOUT_UNLIMITED,
     .window        = scan_window,
     .interval      = BLE_GAP_SCAN_INTERVAL_MAX,
     .channel_mask  = {0}, /* Scanning on all the primary channels */
     .filter_policy = BLE_GAP_SCAN_FP_ACCEPT_ALL
    };

  /* Enable the BLE Stack */
  sd_ble_enable(...);

  /* Start scanning */
  sd_ble_gap_scan_start(&scan_params, &scan_buffer);

  [...]
}
```

API for removing a Vendor Specific base UUID

Using `sd_ble_uuid_vs_remove()`, the application can now remove a Vendor Specific base UUID that has been added with `sd_ble_uuid_vs_add()`. This allows the application to reuse memory allocated for Vendor Specific base UUIDs. The application must provide a pointer to the UUID type to be removed as an input parameter to `sd_ble_uuid_vs_remove()`. The UUID type must not be in use by the ATT Server. A limitation with the current implementation is that the input parameter can only point to `BLE_UUID_TYPE_UNKNOWN` or the last added UUID type.

API to enable or disable extended RC calibration

Extended RC calibration is a new SoftDevice feature that performs additional RC oscillator drift detection and calibration when the SoftDevice is acting as a peripheral and the RC oscillator is used as the SoftDevice clock source. The extended RC calibration is performed in addition to the periodic calibration which is configured when calling `sd_softdevice_enable()`. If using only peripheral connections, the periodic calibration can then be configured with a much longer interval because the peripheral can detect and adjust automatically to clock drift and calibrate when required.

The extended RC calibration is enabled by default. The option `BLE_COMMON_OPT_EXTENDED_RC_CAL` is added to the BLE option API, allowing the application to enable or disable this feature. When using this API, set `ble_common_opt_t::extended_rc_cal::enable` to '1' to enable, or to '0' to disable.

API to get the advertiser Bluetooth device address

A new API `sd_ble_gap_adv_addr_get()` enables the application to get the local Bluetooth device address that is used by the advertiser. The application must provide the advertising handle of the advertiser for the `adv_handle` input parameter, and a pointer to an address structure `p_addr` to be used as the output parameter. The function may only be called when advertising is enabled.

Note: If privacy is enabled, the SoftDevice will generate a new private address every `ble_gap_privacy_params_t::private_addr_cycle_s`, which is configured when calling `sd_ble_gap_privacy_set()`. Depending on when sd_ble_gap_adv_addr_get() is called, the returned address may not be the address that is currently used by the advertiser.

Hardware resource usage API

The API now contains new macros to inform the application about the hardware resources used by the SoftDevice.

- The macro `__NRF_NVIC_SD_IRQ_PRIOS` indicates the interrupt priority levels used by the SoftDevice.
- The macro `__NRF_NVIC_APP_IRQ_PRIOS` indicates the interrupt priority levels available to the application.
- The macros `NRF_SOC_SD_PPI_CHANNELS_SD_ENABLED_MSK` and `NRF_SOC_SD_PPI_CHANNELS_SD_DISABLED_MSK` can be used to identify the PPI channels reserved by the SoftDevice when the SoftDevice is enabled or disabled respectively.
- The macros `NRF_SOC_APP_PPI_CHANNELS_SD_ENABLED_MSK` and `NRF_SOC_APP_PPI_CHANNELS_SD_DISABLED_MSK` can be used to identify the PPI channels available to the application when the SoftDevice is enabled or disabled respectively.
- The macros `NRF_SOC_SD_PPI_GROUPS_SD_ENABLED_MSK` and `NRF_SOC_SD_PPI_GROUPS_SD_DISABLED_MSK` can be used to identify the PPI groups reserved by the SoftDevice when the SoftDevice is enabled or disabled respectively.
- The macros `NRF_SOC_APP_PPI_GROUPS_SD_ENABLED_MSK` and `NRF_SOC_APP_PPI_GROUPS_SD_DISABLED_MSK` can be used to identify the PPI groups available to the application when the SoftDevice is enabled or disabled respectively.

Other additions to the API

- The macro SD_VARIANT_ID indicates the SoftDevice variant.
- The macro SD_FLASH_SIZE indicates the amount of flash memory used by the SoftDevice.

s132_nrf52_6.0.0

This section describes how to migrate to s132_nrf52_6.0.0 from s132_nrf52_5.1.0.

Notes:

- s132_nrf52_6.0.0 has changed the API compared to s132_nrf52_5.1.0 which requires applications to be recompiled.
- s132_nrf52_6.0.0 includes some features that are not Bluetooth qualified. For more information, see the release notes.

New functionality

Quality of Service (QoS) channel survey

This feature provides measurements of the energy levels on the Bluetooth Low Energy channels to the application. The application can use this information to determine the noise floor on a per channel basis and set an adapted channel map to avoid busy channels.

When the feature is enabled, BLE_GAP_EVT_QOS_CHANNEL_SURVEY_REPORT events will periodically report the measured energy levels for each channel. The channel energy is reported in ble_gap_evt_qos_channel_survey_report_t::channel_energy[BLE_GAP_CHANNEL_COUNT], indexed by the Channel Index. The SoftDevice will attempt to measure energy levels and deliver reports with the average interval specified in interval_us.

Note: To make the channel survey feature available to the application, ble_gap_cfg_role_count_t::qos_channel_survey_role_available must be set. This is done using the sd_ble_cfg_set() API.

The event structures for BLE_GAP_EVT_RSSI_CHANGED and BLE_GAP_EVT_ADV_REPORT have been changed to provide the application the channel number for reported Received Signal Strength Indication (RSSI) measurements. For more information, see Updated RSSI API in the Required changes section.

API Updates

- A new Boolean flag, ble_gap_cfg_role_count_t::qos_channel_survey_role_available, must be set in the SoftDevice role configuration API to make the channel survey available for the application.

- Two new SV calls have been added to start and stop the channel survey:
    - `sd_ble_gap_qos_channel_survey_start()`
    - `sd_ble_gap_qos_channel_survey_stop()`

Usage

```
/* Make Channel Survey feature available to the application */
ble_cfg_t cfg;
cfg.role_count.qos_channel_survey_role_available = 1;
sd_ble_cfg_set(..., &cfg, ...);
```

```
/* Start receiving channel survey continuously. */
uint32_t errcode;
errcode = sd_ble_gap_qos_channel_survey_start(BLE_GAP_QOS_CHANNEL_SURVEY_INTERVAL_CONTINUOUS);
```

```
int8_t rssi;
/* A new measurement is ready. */
case BLE_GAP_EVT_QOS_CHANNEL_SURVEY_REPORT:
{
  for (i = 0; i < BLE_GAP_CHANNEL_COUNT; i++)
  {
     rssi = p_ble_evt->evt.gap_evt.params.qos_channel_survey_report.channel_energy[i];
  }
}
```

```
/* Stop receiving channel survey. */
errcode = sd_ble_gap_qos_channel_survey_stop()
```

Advertising Extensions

The LE Advertising Extensions feature has limited support in this SoftDevice that can be enabled with the new advertiser and scanner API. The feature may not function as specified, and may contain issues. For more information, see the release notes.

Extended Advertiser

Extended advertising can be enabled by assigning an `_EXTENDED_` advertising type to the `ble_gap_adv_params_t::properties::type`.

The extended advertising types are:

`BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UNDIRECTED`
`BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_DIRECTED`
`BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_SCANNABLE_UNDIRECTED`
`BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_SCANNABLE_DIRECTED`
`BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_NONSCANNABLE_UNDIRECTED`
`BLE_GAP_ADV_TYPE_EXTENDED_NONCONNECTABLE_NONSCANNABLE_DIRECTED`

New parameters in the API that are relevant for extended advertising:

- `ble_gap_adv_params_t::properties::anonymous`
  - If this flag is set to 1, the advertiser's address will be omitted from all PDUs. This is only available for extended advertising event types.
- `ble_gap_adv_params_t::primary_phy`
  - Indicates the PHY on which the primary advertising channel packets are transmitted.
  - For extended advertising event types, this can be set to `BLE_GAP_PHY_AUTO`, `BLE_GAP_PHY_1MBIT`, or `BLE_GAP_PHY_CODED` if supported by the SoftDevice.
- `ble_gap_adv_params_t::secondary_phy`
  - Indicates the PHY on which the auxiliary PDUs will be sent.
  - Can be set to `BLE_GAP_PHY_AUTO`, `BLE_GAP_PHY_1MBPS`, `BLE_GAP_PHY_2MBPS`, or `BLE_GAP_PHY_CODED` if supported by the SoftDevice.
- `ble_gap_adv_params_t::set_id`
  - This value is used as the Advertising Set ID in the AdvDataInfo field of the PDU.

Extended Scanner

Scanning of extended advertising PDUs can be enabled by setting the `ble_gap_scan_params_t::extended` flag to 1 for the scan parameters provided to `sd_ble_gap_scan_start()`. If set to 1, both legacy and extended advertising PDUs will be scanned. If the flag is set to 0, all extended advertising PDUs will be ignored by the scanner. Correspondingly, to connect to a peer that is advertising with extended advertising PDUs, set the `ble_gap_scan_params_t::extended` flag to 1 for the scan parameters provided to `sd_ble_gap_connect()`.

New parameters in the API that are relevant for extended scanning:

- `ble_gap_scan_params_t::report_incomplete_evts`
  - This option is currently not supported.
- `ble_gap_evt_adv_report_t::type::extended_pdu`
  - Will be set to 1 if an extended advertising set is received.
- `ble_gap_evt_adv_report_t::tx_power`
  - The transmit power reported by the advertising in the last packet header received. The TX power field is present only in some extended advertising PDUs.
- `ble_gap_evt_adv_report_t::aux_pointer`
  - The offset and PHY of the next advertising packet in this extended advertising set.
  - This field will only be set if `ble_gap_evt_adv_report_t::type::status` is set to `BLE_GAP_ADV_DATA_STATUS_INCOMPLETE_MORE_DATA`.
- `ble_gap_evt_adv_report_t::set_id`
  - Set ID of the received advertising data. This is only present in some extended advertising PDUs.
- `ble_gap_evt_adv_report_t::data_id`
  - Data ID of the received advertising data. This is only present in some extended advertising PDUs.

Write to SoftDevice protected registers

A new API, `sd_protected_register_write()`, has been added to give the application the possibility to write to a register that is write-protected by the SoftDevice. A write-protected peripheral shall only be accessed through the SoftDevice API when the SoftDevice is enabled.

The new API supports writing to the Block Protection (BPROT) peripheral. The application can use `sd_protected_register_write()` instead of `sd_flash_protect()` to set the flash protection configuration registers.

Usage

```
/* Old API: */
errcode = sd_flash_protect(value0, value1, value2, value3)

/* New API: */
errcode = sd_protected_register_write(&(NRF_BPROT->CONFIG0), value0)
```

```
errcode = sd_protected_register_write(&(NRF_BPROT->CONFIG1), value1)
errcode = sd_protected_register_write(&(NRF_BPROT->CONFIG2), value2)
errcode = sd_protected_register_write(&(NRF_BPROT->CONFIG3), value3)
```

Required changes

Updated advertiser API

`sd_ble_gap_adv_data_set()` has been removed.

A new API, `sd_ble_gap_adv_set_configure()`, has been added with the following functionalities:

- Configuring and updating the advertising parameters of an advertising set.
- Setting, clearing, or updating advertising and scan response data.

  Note: The advertising data must be kept alive in memory until advertising is terminated. Not doing so will lead to undefined behavior. Note: Updating advertising data while advertising can only be done by providing new advertising data buffers.

Configuring and updating an advertising set

Advertising Set is a term introduced in Bluetooth Core Specification v5.0.

Each advertising set is identified by an advertising handle. To configure a new advertising set and obtain a new advertising handle, `sd_ble_gap_adv_set_configure()` should be called with a pointer `p_adv_handle` pointing to an advertising handle set to `BLE_GAP_ADV_SET_HANDLE_NOT_SET`.

To update an existing advertising set, `sd_ble_gap_adv_set_configure()` should be called with a previously configured advertising handle.

Note: Currently only one advertising set can be configured in the SoftDevice.

Configuring advertising parameters for an advertising set

Setting advertising parameters has been moved from `sd_ble_gap_adv_start()` to `sd_ble_gap_adv_set_configure()`.

The content of `ble_gap_adv_params_t` has changed:

- `ble_gap_adv_params_t::type` has been removed.
- A new parameter, `properties`, of the new type `ble_gap_adv_properties_t` has been added.
  - The advertising type must now be set through `ble_gap_adv_properties_t::type`.

- The advertising type definitions (BLE_GAP_ADV_TYPES) have changed, and new types have been added. The mapping from old to new advertising types is shown below. These advertising types are referred to as *legacy* advertising types:
    - `type = BLE_GAP_ADV_TYPE_ADV_IND -> properties.type = BLE_GAP_ADV_TYPE_CONNECTABLE_SCANNABLE_UNDIRECTED`
    - `type = BLE_GAP_ADV_TYPE_ADV_DIRECT_IND -> properties.type = BLE_GAP_ADV_TYPE_CONNECTABLE_NONSCANNABLE_DIRECTED_HIGH_DUTY_CYCLE` or `BLE_GAP_ADV_TYPE_CONNECTABLE_NONSCANNABLE_DIRECTED`
    - `type = BLE_GAP_ADV_TYPE_ADV_SCAN_IND -> properties.type = BLE_GAP_ADV_TYPE_NONCONNECTABLE_SCANNABLE_UNDIRECTED`
    - `type = BLE_GAP_ADV_TYPE_ADV_NONCONN_IND -> properties.type = BLE_GAP_ADV_TYPE_NONCONNECTABLE_NONSCANNABLE_UNDIRECTED`
- `ble_gap_adv_params_t::fp` has been renamed `ble_gap_adv_params_t::filter_policy`.
- `ble_gap_adv_params_t::timeout` has been renamed `ble_gap_adv_params_t::duration` and is now measured in 10 ms units.
- `ble_gap_adv_params_t::channel_mask` type has been changed from `ble_gap_adv_ch_mask_t` to the new type `ble_gap_ch_mask_t`.
    - Note: At least one of the primary channels that is channel index 37-39 must be set to 0.
    - Note: Masking away secondary channels is currently not supported.
    - The mapping from old type `ble_gap_adv_ch_mask_t` to the new type `ble_gap_ch_mask_t` is shown below:
        - `channel_mask.ch_37_off = 1` -> `channel_mask = 0x2000000000`
        - `channel_mask.ch_38_off = 1` -> `channel_mask = 0x4000000000`
        - `channel_mask.ch_39_off = 1` -> `channel_mask = 0x8000000000`
- `ble_gap_adv_params_t` has several new parameters:
    - `max_adv_evts` has been added to allow the application to advertise for a given number of advertising events.
    - `scan_req_notification` flag has been added to give the application the possibility to receive events of type `ble_gap_evt_scan_req_report_t`. This replaces `BLE_GAP_OPT_SCAN_REQ_REPORT`.
    - `primary_phy` and `secondary_phy` allow the application to select PHYs for primary and secondary advertising channels.
        - `primary_phy` should be set to `BLE_GAP_PHY_AUTO` or `BLE_GAP_PHY_1MBPS` for legacy advertising types. For extended advertising types, it should be set to `BLE_GAP_PHY_1MBPS` or `BLE_GAP_PHY_CODED` if supported by the SoftDevice.
        - `secondary_phy` can be ignored for legacy advertising. For extended advertising types, it should be set to `BLE_GAP_PHY_1MBPS, BLE_GAP_PHY_2MBPS,` or `BLE_GAP_PHY_CODED` if supported by the SoftDevice.
    - `set_id` has been added to allow the application to choose the set ID of an extended advertiser.

Other Advertising API changes

- `BLE_GAP_TIMEOUT_SRC_ADVERTISING` has been removed.
    - A new event, `BLE_GAP_EVT_ADVERTISING_SET_TERMINATED` with structure `ble_gap_evt_adv_set_terminated_t`, has been introduced to let the application know when and why an advertising set has terminated.
- A new configuration parameter, `ble_gap_cfg_role_count_t::adv_set_count`, has been introduced to set the maximum number of advertising sets. Note: The maximum number of supported advertising sets is `BLE_GAP_ADV_SET_COUNT_MAX`.
- `BLE_GAP_ADV_MAX_SIZE` has been replaced with `BLE_GAP_ADV_SET_DATA_SIZE_MAX.`
- `ble_gap_evt_connected_t` now includes `adv_handle` and `adv_data` of the new type `ble_gap_adv_data_t`. These are set when the device connects as a peripheral.
- `ble_gap_evt_scan_req_report_t` now includes `adv_handle`.
- `BLE_GAP_OPT_SCAN_REQ_REPORT` has been removed.
- `BLE_GAP_ADV_TIMEOUT_LIMITED_MAX` has been changed from 180 to 18000 as `sd_ble_gap_adv_params_t::duration` is now measured in 10 ms units.

Usage

```
static uint8_t raw_adv_data_buffer1[BLE_GAP_ADV_SET_DATA_SIZE_MAX];
static uint8_t raw_scan_rsp_data_buffer1[BLE_GAP_ADV_SET_DATA_SIZE_MAX];
static ble_gap_adv_data_t adv_data1 = {.adv_data.p_data      = raw_adv_data_buffer1,      .adv_data.len      =
sizeof(raw_adv_data_buffer1),
                                       .scan_rsp_data.p_data = raw_scan_rsp_data_buffer1, .scan_rsp_data.len =
sizeof(raw_scan_rsp_data_buffer1)};

/* A second advertising data buffer for later updating advertising data while advertising */
static uint8_t raw_adv_data_buffer2[BLE_GAP_ADV_SET_DATA_SIZE_MAX];
static uint8_t raw_scan_rsp_data_buffer2[BLE_GAP_ADV_SET_DATA_SIZE_MAX];
static ble_gap_adv_data_t adv_data2 = {.adv_data.p_data      = raw_adv_data_buffer2,      .adv_data.len      =
sizeof(raw_adv_data_buffer2),
                                       .scan_rsp_data.p_data = raw_scan_rsp_data_buffer2, .scan_rsp_data.len =
sizeof(raw_scan_rsp_data_buffer2)};

int main(void)
{
  uint8_t adv_handle = BLE_GAP_ADV_SET_HANDLE_NOT_SET;
  ble_gap_adv_params_t adv_params = {.properties={.type=BLE_GAP_ADV_TYPE_CONNECTABLE_SCANNABLE_UNDIRECTED},
                                     .interval              = BLE_GAP_ADV_INTERVAL_MAX,
                                     .duration              = BLE_GAP_ADV_TIMEOUT_LIMITED_MAX,
                                     .channel_mask          = {0}, /* Advertising on all the primary channels */
                                     .max_adv_evts          = 0,
                                     .filter_policy         = BLE_GAP_ADV_FP_ANY,
                                     .primary_phy           = BLE_GAP_PHY_AUTO,
                                     .scan_req_notification = 1
                                    };
  /* Enable the BLE Stack */
  sd_ble_enable(...);

  [...]
  sd_ble_gap_adv_set_configure(&adv_handle, &adv_data1, &adv_params);
```

```
    /* Start advertising */
    sd_ble_gap_adv_start(adv_handle, BLE_CONN_CFG_TAG_DEFAULT);

    [...]
    /* Update advertising data while advertising */
    sd_ble_gap_adv_set_configure(&adv_handle, &adv_data2, NULL);

    [...]
    /* Stop advertising */
    sd_ble_gap_adv_stop(adv_handle);

    [...]
  }
```

Updated scanner API

The scanner API has been updated. The changes are as follows:

- `ble_gap_scan_params_t` has been changed:
  - A new flag, `extended`, has been added. If set to 1, the scanner will receive both legacy advertising packets and extended advertising packets. If set to 0, the extended advertising packets will be ignored.
  - The Observer channel map for primary advertising channels can be set through a new parameter `ble_gap_scan_params_t::channel_mask`. The parameter type `ble_gap_ch_mask_t` is the same as is used for setting advertiser channel map.
  - `use_whitelist` and `adv_dir_report` have been combined into `filter_policy`. See `BLE_GAP_SCAN_FILTER_POLICIES` for valid policies.
  - `scan_phys` has been added to let the application decide on which PHYs the scanner should receive packets. Set to `BLE_GAP_PHY_AUTO` orBLE_GAP_PHY_1MBPS if extended scanning is disabled.
  - `timeout` is now measured in 10 ms units.
- `sd_ble_gap_scan_start()` has a new input parameter, `p_adv_report_buffer`, which takes a pointer to an advertising report buffer that must be kept alive until the scanner is stopped. The minimum buffer size is either `BLE_GAP_SCAN_BUFFER_MIN` or `BLE_GAP_SCAN_BUFFER_EXTENDED_MIN` when extended scanning is enabled.
- When the application receives a `ble_gap_adv_report_t`, it must now resume scanning by calling `sd_ble_gap_scan_start()`.
- ble_gap_evt_adv_report_t has been updated:
  - `ble_gap_evt_adv_report_t::type` has been redefined from `uint8_t` to `ble_gap_adv_report_type_t`.
  - `scan_rsp` flag has been removed. It is now included in `ble_gap_adv_report_type_t::scan_response`.
  - `data` and `dlen` have been replaced with `data` of type `ble_data_t`.
  - New fields have been added: and `aux_pointer`.
- `ble_gap_evt_timeout_t` now includes `adv_report_buffer` which is set when the scanner times out.
- `BLE_GAP_SCAN_INTERVAL_MAX` and `BLE_GAP_SCAN_WINDOW_MAX` have been increased from 0x4000 to 0xFFFF.

- `BLE_GAP_SCAN_TIMEOUT_MAX` has been removed.

Usage

```
static uint8_t raw_scan_buffer[BLE_GAP_SCAN_BUFFER_MIN];
static ble_data_t scan_buffer = {.p_data = raw_scan_buffer, .len = sizeof(raw_scan_buffer)};

void on_ble_evt(const ble_evt_t * p_evt)
{
  if (p_ble_evt->header.evt_id == BLE_GAP_EVT_ADV_REPORT)
  {
    ble_gap_evt_adv_report_t * p_report = &p_ble_evt->evt.gap_evt.params.adv_report;

    /* Read out data*/
    [...]

    /* Continue scanning. */
    sd_ble_gap_scan_start(NULL, &scan_buffer);
  }
}

int main(void)
{
  ble_gap_scan_params_t scan_params= {.extended      = 0,
                                       .scan_phys     = BLE_GAP_PHY_AUTO,
                                      .timeout       = BLE_GAP_SCAN_TIMEOUT_UNLIMITED, /* Unlimited scanning */
                                      .interval      = BLE_GAP_SCAN_INTERVAL_MAX,
                                      .channel_mask  = {0}, /* Scanning on all the primary channels */
                                      .filter_policy = BLE_GAP_SCAN_FP_ACCEPT_ALL
                                     };
  /* Enable the BLE Stack */
```

```
    sd_ble_enable(...);

    /* Start scanning */
    sd_ble_gap_scan_start(&scan_params, &scan_buffer);

    [...]
  }
```

Updated RSSI API

The RSSI API has been changed so that the SoftDevice can provide the application with the channel index on which the reported RSSI measurements are made.

- `sd_ble_gap_rssi_get()` takes an additional parameter `p_ch_index`. For this parameter, provide a pointer to a location where the channel index for the RSSI measurement should be stored.
- The event structure for the `BLE_GAP_EVT_RSSI_CHANGED` event has a new parameter `ble_gap_evt_rssi_changed_t::ch_index`. This is the Data Channel Index (0-36) on which the RSSI is measured.
- The event structure for the `BLE_GAP_EVT_ADV_REPORT` event has a new parameter `ble_gap_evt_adv_report_t::ch_index`. This is the Channel Index (0-39) on which the last advertising packet is received. The corresponding measured RSSI for this packet can be read from `ble_gap_evt_adv_report_t::rssi`.

TX power API

The TX power API now supports setting individual transmit power for each link or role.

- `sd_ble_gap_tx_power_set()` takes two new parameters, `role` and `handle`, in addition to `tx_power`. For available roles and TX power values, see ble_gap.h.

Updated Flash API

`sd_flash_write()` now triggers a HardFault if the application tries to write to a protected page. `NRF_ERROR_FORBIDDEN` is returned if the application tries to write to a page outside application flash area.

`sd_flash_page_erase()` now triggers a HardFault if the application tries to erase a protected page. `NRF_ERROR_FORBIDDEN` is returned if the application tries to erase a page outside application flash area.

s132_nrf52_5.0.0

This section describes how to migrate to s132_nrf52_5.0.0 from s132_nrf52_4.0.3.

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 **0x23000**.

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

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

API renaming and updates

Some APIs are renamed or removed. Applications that use the old API names must be updated:

- The timeout source `BLE_GAP_TIMEOUT_SRC_SECURITY_REQUEST` has been removed. Use the existing `BLE_GAP_EVT_AUTH_STATUS {auth_status: BLE_GAP_SEC_STATUS_TIMEOUT}` instead.
- `BLE_GAP_ADV_NONCON_INTERVAL_MIN` has been removed because the lower limit for the advertising interval for non-connectable advertisement has been lowered to `BLE_GAP_ADV_INTERVAL_MIN`.
- The compatibility mode `BLE_GAP_OPT_COMPAT_MODE_2` is removed because the SoftDevice now accepts overlapping peer-initiated Link Layer control procedures as a slave.
- `NRF_ERROR_BUSY` will no longer be returned by `sd_ble_gap_adv_start()`, `sd_ble_gap_scan_start()`, `sd_ble_gap_authenticate()` and `sd_ble_gap_connect()`.
- `NRF_ERROR_BUSY` can now be returned when calling `sd_ble_user_mem_reply()`, `sd_ble_gatts_rw_authorize_reply()` or `sd_ble_gap_sec_params_reply()`.
- `NRF_CLOCK_LF_XTAL_ACCURACY` renamed to `NRF_CLOCK_LF_ACCURACY`
- `nrf_clock_lf_cfg_t` struct member `xtal_accuracy` renamed to `accuracy`.
- A new event `BLE_GAP_EVT_PHY_UPDATE_REQUEST` has been added. The application must check for this event and respond to it by calling the SV call `sd_ble_gap_phy_update()`. For more information, please refer to the 2 Mbps PHY support section in New functionality.

RC Oscillator accuracy

The RC oscillator accuracy can now be set to any of the defined `NRF_CLOCK_LF_ACCURACY` values and there is no default value anymore. In other words, the `nrf_clock_lf_cfg_t::accuracy` parameter now has the same functionality when used with the RCOSC clock source as with the XTAL clock source. The RC oscillator accuracy should be set to a value appropriate for the chip.

New functionality

2 Mbps PHY support

This SoftDevice supports 2 Mbps PHY data transmission for already established connections. Either the application or the peer can request switching to 2 Mbps PHY in order to achieve higher throughput. Both sides need to agree on the PHYs before a PHY change can occur. The application has to respond to the PHY Update procedure when that is initiated by the peer, otherwise the link will be disconnected. This makes it necessary for the application to pull a new event: `BLE_GAP_EVT_PHY_UPDATE_REQUEST`. Another event, `BLE_GAP_EVT_PHY_UPDATE,` may be raised when a PHY Update procedure is completed but the application is not required to take any actions for this event.

API Updates

- A new SV call, `sd_ble_gap_phy_update()`, has been added to request the controller to attempt to change to a new PHY, or to respond to a peer-initiated PHY Update procedure.

Usage

```
ble_gap_phys_t phys = {BLE_GAP_PHY_2MBPS, BLE_GAP_PHY_2MBPS};
sd_ble_gap_phy_update(conn_handle, &phys);
```

- A new event, `BLE_GAP_EVT_PHY_UPDATE_REQUEST`, has been added to notify the application that the peer has initiated a PHY Update procedure, to which the application must respond with its PHY preferences.
- A new event, `BLE_GAP_EVT_PHY_UPDATE`, has been added to notify the application that a self-initiated or peer-initiated PHY Update procedure has been completed.

Usage

```
case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
{
  /* The PHYs requested by the peer can be read from the event parameters: p_ble_evt->evt.gap_evt.params.
phy_update_request.peer_preferred_phys.
   * Note that the peer's TX correponds to our RX and vice versa. */
```

```
    /* Allow SoftDevice to choose PHY Update Procedure parameters automatically. */
    ble_gap_phys_t phys = {BLE_GAP_PHY_AUTO, BLE_GAP_PHY_AUTO};
    sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
    break;
  }
  case BLE_GAP_EVT_PHY_UPDATE:
  {
    if (p_ble_evt->evt.gap_evt.params.phy_update.status == BLE_HCI_STATUS_CODE_SUCCESS)
    {
      /* PHY Update Procedure completed, see p_ble_evt->evt.gap_evt.params.phy_update.tx_phy and p_ble_evt->evt.
gap_evt.params.phy_update.rx_phy for the currently active PHYs of the link. */
    }
    break;
  }
```

Connection-Oriented Channels in LE Credit Based Flow Control Mode

The SoftDevice now supports Connection-Oriented Channels in the LE Credit Based Flow Control Mode. To be able to use this feature, the application has to set an L2CAP connection configuration using the configuration API `sd_ble_cfg_set()` as shown below.

Usage

Setting L2CAP Connection Configuration

```
/* Set L2CAP Connection Configuration for connection identified by coc_conn_cfg_tag */
ble_cfg_t cfg;

memset(&cfg, 0, sizeof(ble_cfg_t));

cfg.conn_cfg.conn_cfg_tag                                                         = coc_conn_cfg_tag;
cfg.conn_cfg.params.l2cap_conn_cfg.rx_mps                         = BLE_L2CAP_MPS_MIN;
```

```
cfg.conn_cfg.params.l2cap_conn_cfg.tx_mps                         = BLE_L2CAP_MPS_MIN;
cfg.conn_cfg.params.l2cap_conn_cfg.rx_queue_size         = 5;
cfg.conn_cfg.params.l2cap_conn_cfg.tx_queue_size         = 5;
cfg.conn_cfg.params.l2cap_conn_cfg.ch_count                 = 1;

sd_ble_cfg_set(BLE_CONN_CFG_L2CAP, &cfg, ...);

[...]

/* Enable the BLE Stack */
sd_ble_enable(...);
```

The usage of some of the SV calls and events related to this feature is explained below. For the complete list of SV calls and events, please refer to the API documentation available in `ble_l2cap.h`.

A new SV call, `sd_ble_l2cap_ch_setup()`, has been added to request the setup of an L2CAP channel, or to respond to a setup request from a peer.

Usage

Creating a new L2CAP Channel

```
uint16_t                                          local_cid = BLE_L2CAP_CID_INVALID;
ble_l2cap_ch_setup_params_t ch_setup_params;

ch_setup_params.le_psm                                = 0x25;
ch_setup_params.rx_params.rx_mtu         = BLE_L2CAP_MTU_MIN;
ch_setup_params.rx_params.rx_mps         = BLE_L2CAP_MPS_MIN;
ch_setup_params.rx_params.sdu_buf         = NULL;
sd_ble_l2cap_ch_setup(conn_handle, &local_cid, &ch_setup_params);
```

Responding to a L2CAP Channel setup request

```
case BLE_L2CAP_EVT_CH_SETUP_REQUEST:
{
  /* An L2CAP channel setup request has been received from the peer. */

  uint16_t                                           local_cid = p_ble_evt->evt.l2cap_evt.local_cid;
  ble_l2cap_ch_setup_params_t ch_setup_params;

  ch_setup_params.le_psm                         = 0x25;
  ch_setup_params.rx_params.rx_mtu         = BLE_L2CAP_MTU_MIN;
  ch_setup_params.rx_params.rx_mps         = BLE_L2CAP_MPS_MIN;
  ch_setup_params.rx_params.sdu_buf = NULL;

  sd_ble_l2cap_ch_setup(p_ble_evt->evt.l2cap_evt.conn_handle, &local_cid, &ch_setup_params);

  break;
}
```

The SV call `sd_ble_l2cap_ch_tx()` can be used to transmit an SDU (Service Data Unit) on an L2CAP channel. The event `BLE_L2CAP_EVT_CH_TX` is generated by the SoftDevice to notify the application that the SDU has been transmitted.

Usage

Transmitting on an L2CAP Channel

```
ble_data_t sdu_to_send;
uint8_t    data[] = "Sample";

sdu_to_send.len    = strlen(data);
sdu_to_send.p_data = data;
```

```
sd_ble_l2cap_ch_tx(conn_handle, local_cid, &sdu_to_send);

[...]
case BLE_L2CAP_EVT_CH_TX:
  /* The SDU is transmitted. */
  break;

[...]
```

The SV call `sd_ble_l2cap_ch_rx()` shall be used to provide the SoftDevice with a buffer to receive an SDU from the peer. The event `BLE_L2CAP_EVT_CH_RX` is generated by the SoftDevice to notify the application that an SDU has been received. The application shall not change the buffer provided to the SoftDevice before receiving the event.

Usage

Receiving on an L2CAP Channel

```
[...]

ble_data_t sdu_buf;
uint8_t    data[150];

sdu_buf.len    = strlen(data);
sdu_buf.p_data = data;

sd_ble_l2cap_ch_rx(conn_handle, local_cid, &sdu_buf));

[...]
case BLE_L2CAP_EVT_CH_RX:
  /* An SDU is received by the SoftDevice from the peer and is available in p_ble_evt->evt.l2cap_evt.params.rx.
```

```
sdu_buf */

  break;

[...]
```

Network Privacy Mode

The SoftDevice now supports the Network Privacy Mode. In Network Privacy Mode, a device will only accept advertising packets from peer devices that contain private addresses.

API Updates

- A new mode, `BLE_GAP_PRIVACY_MODE_NETWORK_PRIVACY`, is added to enable Network Privacy Mode.
- A new characteristic, `BLE_UUID_GAP_CHARACTERISTIC_RPA_ONLY` (RPA = Resolvable Private Address), is defined to let the application add this characteristic to the attribute database.

Usage

Set the privacy settings to network privacy with random private resolvable address:

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

Unique string to identify a SoftDevice

The SoftDevice Information Structure now also contains a string, namely the SoftDevice unique string, that can be used to uniquely identify a version of the SoftDevice (applies also for alpha releases).

A new define `SD_UNIQUE_STR_ADDR_GET` has been added to retrieve the address of the SoftDevice unique string. The defines `SD_UNIQUE_STR_SIZE` and `SD_UNIQUE_STR_OFFSET` define the size of the string and its offset relative to the SoftDevice base address respectively.

Usage

Fetching the SoftDevice unique string

```
/* Declare a character array that is twice the length of the SoftDevice unique string.
 * This will be used to store the hexadecimal representation of the SoftDevice unique string. */
char str[SD_UNIQUE_STR_SIZE * 2];
/* Fetch the address of the SoftDevice unique string. */
const uint8_t * const p_unique_str = SD_UNIQUE_STR_ADDR_GET(MBR_SIZE);

/* Read the SoftDevice unique string into the character array, converting it into hexadecimal notation. */
for (uint8_t i = 0; i < SD_UNIQUE_STR_SIZE; i++)
{
  sprintf(&str[i * 2], "%02x", p_unique_str[i]);
}
/* The SoftDevice unique string is now available in the character array named str. */
```

Other API additions and changes

- The status code `BLE_HCI_STATUS_CODE_LMP_ERROR_TRANSACTION_COLLISION` indicates that there has been an illegal collision of LL Control PDUs on air.
- A new MBR command `SD_MBR_COMMAND_IRQ_FORWARD_ADDRESS_SET` has been added to forward all interrupts to another base address.
- New defines for minimum and maximum values of authenticated payload timeout have been added. See `BLE_GAP_AUTH_PAYLOAD_TIMEOUT`.
- A flag `lesc` is added to the `ble_gap_evt_auth_status_t` struct, indicating whether an authentication procedure resulted in an LE Secure Connection.
- The SoftDevice will no longer return `NRF_ERROR_BUSY` on `sd_ble_gap_conn_param_update()` unless the procedure is already in progress.
- The new definitions `NRF_RADIO_MAX_EXTENSION_PROCESSING_TIME_US` and `NRF_RADIO_MIN_EXTENSION_MARGIN_US` define timing constraints the application must take into account when using `NRF_RADIO_SIGNAL_CALLBACK_ACTION_EXTEND` with the Radio Timeslot API.

s132_nrf52_4.0.3

This section describes how to migrate to s132_nrf52_4.0.3 from s132_nrf52_3.0.0.

Required changes

SoftDevice RAM usage

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

New configuration API

Configuration parameters passed to `sd_ble_enable()` have been moved to the SoftDevice configuration API.

API updates

- A new SV call `sd_ble_cfg_set()` is added to set the configuration. This API can be called many times to configure different parts of the BLE stack. All configurations are optional. Configuration parameters not set by this API will take their default values.
- The SV call parameter `ble_enable_params_t * p_ble_enable_params` is removed from `sd_ble_enable()`. The SV call `sd_ble_cfg_set()` must be used instead. The parameters of this call are given in the following table:

Old API: `ble_enable_params_t` member	New API: `cfg_id` in `sd_ble_cfg_set()`
`common_enable_params.vs_uuid_count`	`BLE_COMMON_CFG_VS_UUID`
`common_enable_params.p_conn_bw_counts`	`BLE_CONN_CFG_GAP` (*)

`gap_enable_params.periph_conn_count`
`gap_enable_params.central_conn_count`
`gap_enable_params.central_sec_count`
	`BLE_GAP_CFG_ROLE_COUNT`

`gap_enable_params.p_device_name`	`BLE_GAP_CFG_DEVICE_NAME`
`gatt_enable_params`	`BLE_CONN_CFG_GATT` (*)
`gatts_enable_params.service_changed`	`BLE_GATTS_CFG_SERVICE_CHANGED`
`gatts_enable_params.attr_tab_size`	`BLE_GATTS_CFG_ATTR_TAB_SIZE`

(*) These configurations can be set per link.

Usage

Example pseudo code to set per link ATT_MTU using the new configuration API:

```
const uint16_t client_rx_mtu = 158;
const uint32_t long_att_conn_cfg_tag = 1;

/* set ATT_MTU for connections identified by long_att_conn_cfg_tag */
ble_cfg_t cfg;
memset(&cfg, 0, sizeof(ble_cfg_t));
cfg.conn_cfg.conn_cfg_tag = long_att_conn_cfg_tag;
cfg.conn_cfg.params.gatt_conn_cfg.att_mtu = client_rx_mtu;
sd_ble_cfg_set(BLE_CONN_CFG_GATT, &cfg, ...);

/* Enable the BLE Stack */
sd_ble_enable(...);

[...]

uint16_t long_att_conn_handle;
/* Establish connection with long_att_conn_cfg_tag */
sd_ble_gap_adv_start(..., long_att_conn_cfg_tag);

[...]

/* Establish connection with BLE_CONN_CFG_TAG_DEFAULT, it will use default ATT_MTU of 23 bytes */
sd_ble_gap_connect(..., BLE_CONN_CFG_TAG_DEFAULT);

[...]
```

```
/* Start ATT_MTU exchange */
sd_ble_gattc_exchange_mtu_request(long_att_conn_handle, client_rx_mtu);
```

BLE bandwidth configuration

The BLE bandwidth configuration and application packet concept has been changed. Previously, the application could specify a bandwidth setting, which would result in a given queue size and a correpsonding given radio time allocated. Now the queue sizes and the allocated radio time have been separated. The application can now configure:

- Event length
- Write without response queue size
- Handle Value Notification queue size

These settings are configurable per link.

Note that now the configured queue sizes are not directly related to on-air bandwidth:

- The application can configure one single packet to be queued in the SoftDevice, but still achieve full throughput if the application can queue packets fast enough during connection events.
- Even if the application configures a large number of packets to be queued, not all of them will be sent during a single connection event if the configured event length is not large enough to send the packets.

API updates

- The `ble_enable_params_t::common_enable_params.p_conn_bw_counts` parameter of the `sd_ble_enable()` SV call is replaced by the `sd_ble_cfg_set()` SV call with `cfg_id` parameter set to `BLE_CONN_CFG_GAP`. The following table shows how the old bandwidth configuration corresponds to the new one for the default ATT_MTU:

Old API: `BLE_CONN_BWS`	New API: `ble_gap_conn_cfg_t::event_length` in `sd_ble_cfg_set()`
`BLE_CONN_BW_LOW`	`BLE_GAP_EVENT_LENGTH_MIN`
`BLE_CONN_BW_MID`	`BLE_GAP_EVENT_LENGTH_DEFAULT`
`BLE_CONN_BW_HIGH`	`6`

  The bandwidth configuration is further described in the SDS.
- The `BLE_COMMON_OPT_CONN_BW` option is removed. Instead, during connection creation, the application should supply the `conn_cfg_tag` defined by the `ble_conn_cfg_t::conn_cfg_tag` parameter in the `sd_ble_cfg_set()` SV call.
- A new parameter `conn_cfg_tag` is added to `sd_ble_gap_adv_start()` and `sd_ble_gap_connect()` SV calls. To create a connection with a default configuration, `BLE_CONN_CFG_TAG_DEFAULT` should be provided in this parameter.
- The `BLE_EVT_TX_COMPLETE` event is split on two events: `BLE_GATTC_EVT_WRITE_CMD_TX_COMPLETE` and `BLE_GATTS_EVT_HVN_TX_COMPLETE`.

- The SV call `sd_ble_tx_packet_count_get()` is removed. Instead, the application can now configure packet counts per link, using the SV call `sd_ble_cfg_set()` with the `cfg_id` parameter set to `BLE_CONN_CFG_GATTC` and `BLE_CONN_CFG_GATTS`.

Usage

Example pseudo code to set configuration that corresponds to the old `BLE_CONN_BW_HIGH` bandwidth configuration both in throughput and packet queueing capability:

```
const uint32_t high_bw_conn_cfg_tag = 1;
ble_cfg_t cfg;

/* configure connections identified by high_bw_conn_cfg_tag */

/* set connection event length */
memset(&cfg, 0, sizeof(ble_cfg_t));
cfg.conn_cfg.conn_cfg_tag = high_bw_conn_cfg_tag;
cfg.conn_cfg.params.gap_conn_cfg.event_length = 6; /* 6 * 1.25 ms = 7.5 ms corresponds to the old
BLE_CONN_BW_HIGH for default ATT_MTU */
cfg.conn_cfg.params.gap_conn_cfg.conn_count = 1;   /* application needs one link with this configuration */
sd_ble_cfg_set(BLE_CONN_CFG_GAP, &cfg, ...);

/* set HVN queue size */
memset(&cfg, 0, sizeof(ble_cfg_t));
cfg.conn_cfg.conn_cfg_tag = high_bw_conn_cfg_tag;
cfg.conn_cfg.params.gatts_conn_cfg.hvn_tx_queue_size = 7; /* application wants to queue 7 HVNs */
sd_ble_cfg_set(BLE_CONN_CFG_GATTS, &cfg, ...);

/* set WRITE_CMD queue size */
memset(&cfg, 0, sizeof(ble_cfg_t));
cfg.conn_cfg.conn_cfg_tag = high_bw_conn_cfg_tag;
cfg.conn_cfg.params.gattc_conn_cfg.write_cmd_tx_queue_size = 0; /* application is not going to send WRITE_CMD,
so set to 0 to save memory */
sd_ble_cfg_set(BLE_CONN_CFG_GATTC, &cfg, ...);
```

```
/* Enable the BLE Stack */
sd_ble_enable(...);

[...]

uint16_t high_bw_conn_handle;
/* Establish connection with high_bw_conn_cfg_tag */
sd_ble_gap_adv_start(..., high_bw_conn_cfg_tag);
```

Data Length Update Procedure

The application now has to respond to the Data Length Update Procedure when initiated by the peer. See the description of the Data Length Update Procedure in the New functionality section for more details.

Required changes:

```
case BLE_GAP_EVT_DATA_LENGTH_UPDATE_REQUEST:
{
  /* Allow SoftDevice to choose Data Length Update Procedure parameters automatically. */
  sd_ble_gap_data_length_update(p_ble_evt->evt.gap_evt.conn_handle, NULL, NULL);
  break;
}
case BLE_GAP_EVT_DATA_LENGTH_UPDATE:
{
  /* Data Length Update Procedure completed, see p_ble_evt->evt.gap_evt.params.data_length_update.
effective_params for negotiated parameters. */
  break;
}
```

Access to `RAM[x].POWER` registers

SoftDevice APIs are updated to provide access to the `RAM[x].POWER` registers instead of the deprecated `RAMON/RAMONB`.

API updates

- `sd_power_ramon_set()` SV call is replaced with `sd_power_ram_power_set()`.
- `sd_power_ramon_clr()` SV call is replaced with `sd_power_ram_power_clr()`.
- `sd_power_ramon_get()` SV call is replaced with `sd_power_ram_power_get()`.

API rename

Some APIs were renamed. Applications that use the old names must be updated.

API updates

- `BLE_EVTS_PTR_ALIGNMENT` is renamed to `BLE_EVT_PTR_ALIGNMENT`.
- `BLE_EVTS_LEN_MAX` is renamed to `BLE_EVT_LEN_MAX`.
- `GATT_MTU_SIZE_DEFAULT` is renamed to `BLE_GATT_ATT_MTU_DEFAULT`.
- The GAP option `BLE_GAP_OPT_COMPAT_MODE` is renamed to `BLE_GAP_OPT_COMPAT_MODE_1`.
- `ble_gap_opt_compat_mode_t` structure is renamed to `ble_gap_opt_compat_mode_1_t`.
- `ble_gap_opt_compat_mode_t::mode_1_enable` structure member is renamed to `ble_gap_opt_compat_mode_1_t::enable`.
- `ble_gap_opt_t::compat_mode` structure member is renamed to `ble_gap_opt_t::compat_mode_1`.

Proprietary L2CAP API removed

The proprietary API for sending and receiving data over L2CAP is removed.

API updates

- The SV calls `sd_ble_l2cap_cid_register()`, `sd_ble_l2cap_cid_unregister()`, and `sd_ble_l2cap_tx()`are removed.
- `BLE_L2CAP_EVT_RX` event is removed.
- The following defines are removed: `BLE_L2CAP_MTU_DEF`, `BLE_L2CAP_CID_INVALID`, `BLE_L2CAP_CID_DYN_BASE`, `BLE_L2CAP_CID_DYN_MAX`.

New functionality

Data Length Update Procedure

The application is given control of the Data Length Update Procedure. The application can initiate the procedure and has to respond when initiated by the peer.

API updates

- A new SV call `sd_ble_gap_data_length_update()` is added to initiate or respond to a Data Length Update Procedure.
- The `BLE_EVT_DATA_LENGTH_CHANGED` event is replaced with `BLE_GAP_EVT_DATA_LENGTH_UPDATE`.
- A new event `BLE_GAP_EVT_DATA_LENGTH_UPDATE_REQUEST` is added to notify that a Data Length Update request has been received. `sd_ble_gap_data_length_update()` must be called by the application after this event has been received to continue the Data Length Update Procedure.
- The GAP option `BLE_GAP_OPT_EXT_LEN` is removed. The `sd_ble_gap_data_length_update()` SV call should be used instead.

Usage

- The Data Length Update Procedure can be initiated locally or by peer device.
- Following is the pseudo code for the case where Data Length Update Procedure is initiated by application:

```
const uint16_t client_rx_mtu = 247;
const uint32_t long_att_conn_cfg_tag = 1;

/* ATT_MTU must be configured first */
ble_cfg_t cfg;
memset(&cfg, 0, sizeof(ble_cfg_t));
cfg.conn_cfg.conn_cfg_tag = long_att_conn_cfg_tag;
cfg.conn_cfg.params.gatt_conn_cfg.att_mtu = client_rx_mtu;
sd_ble_cfg_set(BLE_CONN_CFG_GATT, &cfg, ...);

/* Enable the BLE Stack */
sd_ble_enable(...);

[...]

uint16_t long_att_conn_handle;
/* Establish connection */
sd_ble_gap_adv_start(..., long_att_conn_cfg_tag);

[...]

/* Start Data Length Update Procedure, can be done without ATT_MTU exchange */
```

```
ble_gap_data_length_params_t params = {
  .max_tx_octets  = client_rx_mtu + 4,
  .max_rx_octets  = client_rx_mtu + 4,
  .max_tx_time_us = BLE_GAP_DATA_LENGTH_AUTO,
  .max_rx_time_us = BLE_GAP_DATA_LENGTH_AUTO
};
sd_ble_gap_data_length_update(long_att_conn_handle, &params, NULL);

[...]

case BLE_GAP_EVT_DATA_LENGTH_UPDATE:
{
  /* Data Length Update Procedure completed, see p_ble_evt->evt.gap_evt.params.data_length_update.
effective_params for negotiated parameters. */
  break;
}
```

New compatibility mode

A new compatibility mode is added to enable interoperability with central devices that may initiate version exchange and feature exchange control procedures in parallel. To enable this mode, use the `sd_ble_opt_set()` SV call with the `opt_id` parameter set to `BLE_GAP_OPT_COMPAT_MODE_2`.

Slave latency configuration

It is now possible to disable and enable slave latency on an active peripheral link. To disable or re-enable slave latency, use the `sd_ble_opt_set()` SV call with the `opt_id` parameter set to `BLE_GAP_OPT_SLAVE_LATENCY_DISABLE`.

Support for high accuracy LFCLK oscillator source

It is now possible to configure the SoftDevice with higher accuracy LFCLK oscillator source. Four new levels are defined:

```
#define NRF_CLOCK_LF_XTAL_ACCURACY_10_PPM  (8) /**< 10 ppm */
#define NRF_CLOCK_LF_XTAL_ACCURACY_5_PPM   (9) /**<  5 ppm */
#define NRF_CLOCK_LF_XTAL_ACCURACY_2_PPM  (10) /**<  2 ppm */
#define NRF_CLOCK_LF_XTAL_ACCURACY_1_PPM  (11) /**<  1 ppm */
```

RC oscillator: "xtal_accuracy" must be configured

In previous versions of the SoftDevice, the `xtal_accuracy` was ignored by the API when RCOSC was selected as the low frequency clock source. The default configuration used was 250 ppm. The RC oscillator accuracy must now be configured by setting `nrf_clock_lf_cfg_t::xtal_accuracy` to `NRF_CLOCK_LF_XTAL_ACCURACY_250_PPM` to maintain the behavior of previous SoftDevices. The only other valid configuration is `NRF_CLOCK_LF_XTAL_ACCURACY_500_PPM`. If the `xtal_accuracy` is set to any value other than 250 ppm or 500 ppm, a default configuration of 500 ppm will be applied.

New power failure levels

It is now possible to configure the SoftDevice with all the new power failure levels introduced in NRF52. Levels that are added:

```
NRF_POWER_THRESHOLD_V17       /**< Set the power failure threshold to 1.7 V. */
NRF_POWER_THRESHOLD_V18       /**< Set the power failure threshold to 1.8 V. */
NRF_POWER_THRESHOLD_V19       /**< Set the power failure threshold to 1.9 V. */
NRF_POWER_THRESHOLD_V20       /**< Set the power failure threshold to 2.0 V. */
NRF_POWER_THRESHOLD_V22       /**< Set the power failure threshold to 2.2 V. */
NRF_POWER_THRESHOLD_V24       /**< Set the power failure threshold to 2.4 V. */
NRF_POWER_THRESHOLD_V26       /**< Set the power failure threshold to 2.6 V. */
NRF_POWER_THRESHOLD_V28       /**< Set the power failure threshold to 2.8 V. */
```

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 parameter 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 `__LINKER_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.
uuid128, 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.