s110_nrf51822 migration document

Introduction to the s110_nrf51822 migration document

This document describes how to migrate to new versions of the s110_nrf51822. The s110_nrf51822 release notes should be read in conjunction with this document.

There is one main section for each new version of the s110_nrf51822. Within each main section,

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

s110_nrf51822_5.1.0

This section describes how to migrate to s110_nrf51822_5.1.0 from s110_nrf51822_5.0.0.

Required changes

No changes are required due to this upgrade.

Note: The *nrf_power_dcdc_mode** API will be deprecated in a future version. See the "Limitations" section in the release notes for further information.

New functionality

No new functionality in this version.

s110_nrf51822_5.0.0

This section describes how to migrate to s110_nrf51822_5.0.0 from s110_nrf51822_4.0.0

Required changes

Lower stack (Link Layer) interrupts are extended by a “CPU Suspend” state during radio activity to improve link integrity. This means lower stack interrupts will **block application and upper stack (Host) processing during a Radio Event** for a time proportional to the number of packets transferred in the event. The application's interrupt latency (both for App High and App Low interrupts) will therefore also increase by a time proportional to the number of packets transferred in the event due to the lower stack preventing the execution of lower priority contexts. Applications relying on low latency interrupts while the radio is active will therefore have to adapt to avoid timing issues.

The impact of this change and the required modifications to the application will highly depend on the application's nature, requirements and behavior. However, in general terms the application should refer to Table 10 (S110 interrupt latency lower stack) in the S110 SoftDevice Specification v1.1 to analyze how radio traffic will impact interrupt latency and incorporate the numbers in the design and implementation of the application's interrupt handlers.

A typical example of the timing potentially being off is using a TIMER or RTC peripheral running with a period shorter than the interrupt latencies described in the table above. Such short period configurations may cause the Interrupt Service Routine to miss ticks if the implementation keeps time by counting on its execution at a fixed frequency. A different timekeeping strategy needs to be used in this case, such as increasing the period (lowering the frequency) or accounting for missed Interrupt Service Routine execution instances by using the relevant information in registers provided by those peripherals.

The following **GATT Characteristic Presentation Format and Namespace definitions:**

- `BLE_GATT_CPF_FORMAT_*`
- `BLE_GATT_CPF_NAMESPACE_*`

have been moved from `ble_types.h` to `ble_gatt.h`.

Two additional definitions have been added to `ble_gatts.h`:

- `BLE_GATTS_FIX_ATTR_LEN_MAX`
- `BLE_GATTS_VAR_ATTR_LEN_MAX`

These macros define the maximum attribute value length for fixed and variable length attributes respectively.

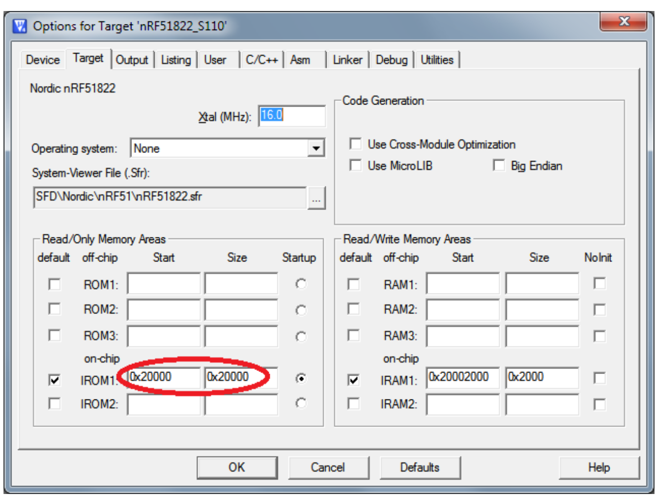
s110_nrf51822_4.0.0

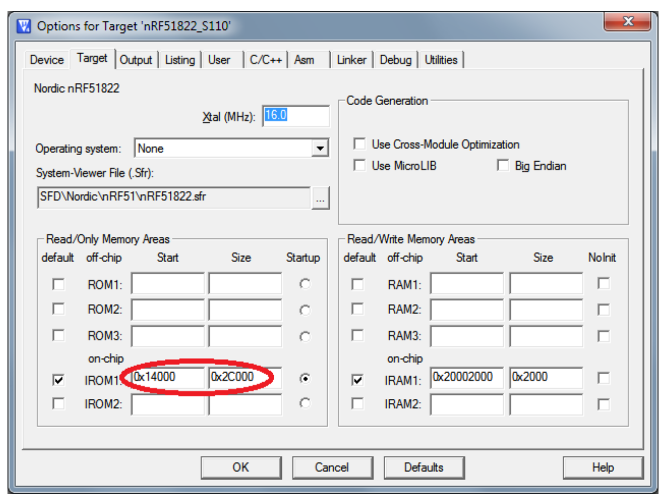
This section describes how to migrate to s110_nrf51822_4.0.0 from s110_nrf51822_3.0.0

Required changes

The flash requirements for the S110 SoftDevice have changed from 128kB to 80kB as of this version. All future releases of the **S110** SoftDevice will require 80kB of flash memory (i.e. CODE_R1_BASE = 0x00014000). To adapt to this new size, applications will need to redefine their base addresses from ~~**`0x00020000`**~~ to **`0x00014000`**.

In Keil, you can achieve this by changing the IROM1 value in the target options:

<table>
<tr><td>From:</td><td>To:</td></tr>
<tr><td>


</td><td>


</td></tr>
</table>

All SuperVisor Calls have been renamed in the 4.0.0 SoftDevice according to the following rules:

- `nrf_*` calls are now `sd_*`
- `ble_*` calls are now `sd_ble_*`
- SVC_* SVC IDs are now SD_*

Application source files that invoke SuperVisor Calls will have to be modified accordingly to match the new naming convention.
When debugging an application the presence of an "`sd_`" prefix now signals the jump to a SoftDevice function in the form of an SVC.

The `sd_power_pof_enable()` function no longer uses a callback. Instead, use `sd_power_pof_enable(true)` to enable power failure detection; the `NRF_EVENT_POWER_FAILURE_WARNING` event will be returned by `sd_event_get()`if a power failure occurs.

UUID conversion functions are now in the SoftDevice. All UUIDs are represented in the S110 SoftDevice as a structure named **`ble_uuid_t`** containing 2 fields (`uuid` and `type`). This representation achieves better speed and memory efficiency than standard 2 and 16 byte arrays and makes it easier to interface with APIs using a unified UUID type. Starting from 4.0.0, the encoding and decoding functions are now inside the BLE stack instead of being in the SDK, and the table population function has changed. To be able to use the new functionality, the application will have to first add all the required 128-bit UUIDs so that they can be referenced later by the ble_uuid_t instances, calling the population function as many times as necessary.

This used to be achieved with the `ble_uuid_vs_assign()` function, which has now been removed in favor of `sd_ble_uuid_vs_add()`:

```
const ble_uuid128_t vs_uuids[] = {{0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC,
0x00, 0x00, 0xFF, 0x17},
{0xA1, 0xB2, 0xC3, 0xD4, 0xE5, 0xF6, 0x07, 0x18, 0x29, 0x3A, 0x4B, 0x5C, 0x00, 0x00, 0x6F, 0x77}};

uint8_t type;
uint32_t i, errcode;
```

~~`errcode = ble_uuid_base_set(NUMELTS(vs_uuids), vs_uuids);`~~

or

~~`errcode = ble_uuid_vs_assign(NUMELTS(vs_uuids), vs_uuids);`~~

```
for(i = 0; i < NUMELTS(vs_uuids); i++)
{
  errcode = sd_ble_uuid_vs_add((ble_uuid128_t const *) &vs_uuids[i], &type);
  ASSERT(errcode == NRF_SUCCESS);
  ASSERT(type == BLE_UUID_TYPE_VENDOR_BEGIN + i);
}
```

Later the code can call the encoding and decoding functions to convert 16 or 128-bit UUIDs into or from ble_uuid_t structures:

```
const ble_uuid128_t raw_sig_uuid = {0xFB, 0x34, 0x9B, 0x5F, 0x80, 0x00, 0x00, 0x80, 0x00, 0x10, 0x00, 0x00,
0x08, 0x82, 0x00, 0x00};
const ble_uuid128_t raw_vs_uuid = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC,
0x34, 0x12, 0xFF, 0x17};

uint8_t raw_output[16];
ble_uuid_t ble_uuid;
uint8_t uuid_len;
```

~~`errcode = ble_uuid_decode(16, raw_sig_uuid.uuid128, &ble_uuid);`~~
`errcode = sd_ble_uuid_decode(16, raw_sig_uuid.uuid128, &ble_uuid);`

~~`errcode = ble_uuid_encode(&ble_uuid, &uuid_len, raw_output);`~~
`errcode = sd_ble_uuid_encode(&ble_uuid, &uuid_len, raw_output);`

```
~~errcode = ble_uuid_decode(16, raw_vs_uuid.uuid128, &ble_uuid);~~
errcode = sd_ble_uuid_decode(16, raw_vs_uuid.uuid128, &ble_uuid);

~~errcode = ble_uuid_encode(&ble_uuid, &uuid_len, raw_output);~~
errcode = sd_ble_uuid_encode(&ble_uuid, &uuid_len, raw_output);
```

Applications can now check directly for characteristic properties in the corresponding bitfield included in the characteristic discovery response:

```
ble_evt_t ble_evt;

~~if((ble_evt.evt.gattc_evt.params.char_disc_rsp.chars[0].properties & 0x4)~~
~~&& (ble_evt.evt.gattc_evt.params.char_disc_rsp.chars[0].properties & 0x10))~~
~~{...}~~

if(ble_evt.evt.gattc_evt.params.char_disc_rsp.chars[0].char_props.write_wo_resp
&& ble_evt.evt.gattc_evt.params.char_disc_rsp.chars[0].char_props.notify)
{...}
```

The characteristic properties types are now shared between client and server and so GATTS service population has changed slightly:

```
ble_gatts_char_md_t char_md;

~~char_md.char_properties.notify = 1;~~
~~char_md.char_properties.broadcast = 1;~~
~~char_md.char_properties.wr_aux = 1;~~

char_md.char_props.notify = 1;
char_md.char_props.broadcast = 1;
char_md.char_ext_props.wr_aux = 1;
```

New functionality

It is now possible to clear the advertising data and/or the scan response data, and the combinations stand today as shown below:

```
sd_ble_gap_adv_data_set(uint8_t const * const p_data, uint8_t dlen, uint8_t const * const p_sr_data,
uint8_t srdlen));


p_data == NULL && dlen == 0      /* adv data unchanged */
p_sr_data == NULL && srdlen == 0 /* scan response data unchanged */
p_data != NULL && dlen == 0      /* adv data cleared (0-length packet) */
p_sr_data != NULL && srdlen == 0 /* scan response data cleared (0-length packet) */
```