NORDIC
SEMICONDUCTOR

Introduction to S110 SoftDevice

nRF51 Series

User Guide v1.0

1 Introduction

This document provides an overview of the S110 SoftDevice including information about the API, how to write applications, and how to handle events coming from the SoftDevice. The S110 SoftDevice can be used to build applications on *Bluetooth*® low energy chips from the nRF51 series.

A SoftDevice is a protocol stack solution that runs in a protected code area with an accompanying protected RAM area. This provides the stack with complete protection from the application when it is running, preventing the application from causing failures inside the stack. The SoftDevice is a precompiled and pre-linked HEX file that is independent from the application and can be programmed separately.

The S110 SoftDevice implements a single mode *Bluetooth* low energy (BLE) stack, and can be used in a Peripheral role or a Broadcaster role.

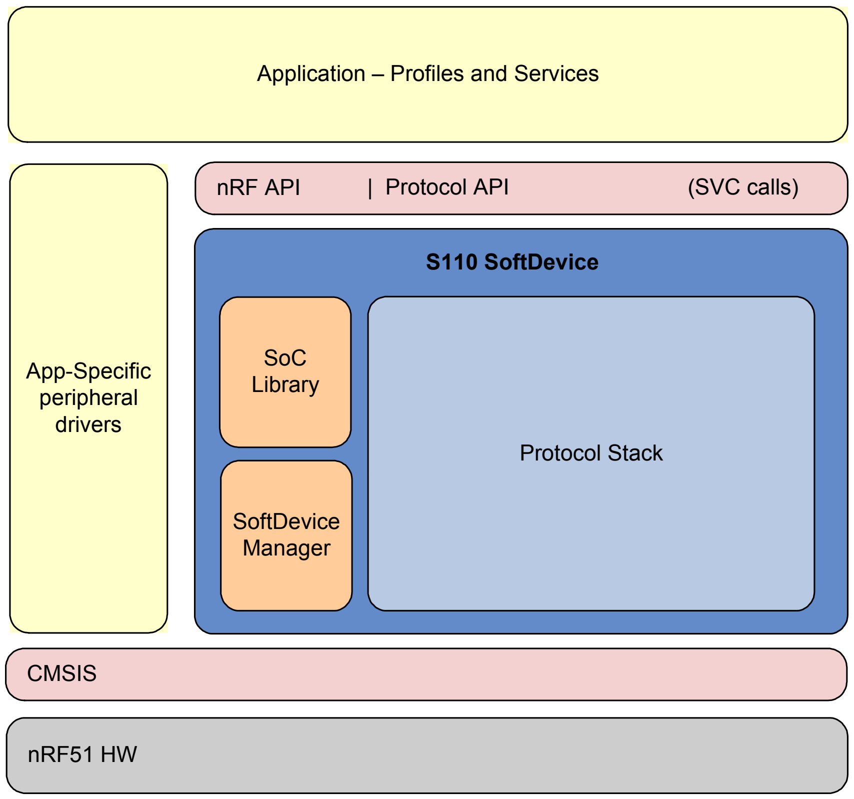

Figure 1 *System on Chip application with the SoftDevice*

1.1 Required reading

It is recommended that the following documents are read before using the S110 SoftDevice:

- *S110 nRF51822 SoftDevice Specification*
- *nRF51822 Product Specification*
- *nRF51 Series Reference Manual*
- *Bluetooth Core Specification* Version 4.0 Volume 3, parts C, F and G, and Volume 6 available at https://www.bluetooth.org.

2 API

The SoftDevice's API is modeled on the functions and procedures defined in the *Bluetooth Core Specification*. All of the API functions are implemented as supervisor calls (SVC). This means that when you call an API function the ARM SVC instruction is executed, with a number as an operand. This causes an exception (in ARM's terms) and the SVC handler inside the SoftDevice to run. The handler determines the API function that was called by looking at the SVC operand.

The number of parameters to any SoftDevice API function is limited to four by the SVC mechanism. Most API functions require more than four parameters, so they usually take one or more pointers to C structures as parameters. Further parameters can be read as elements in those structures.

The API is defined in a set of header files which are divided based on which part of the *Bluetooth Core Specification* they relate to (for example GAP, GATT Client, GATT Server, and so on). Each header file contains declarations for API methods, data structures, error codes, and similar which are related to the corresponding part of the *Bluetooth Core Specification*. For example, a GAP event (such as BLE_GAP_EVT_CONNECTED) will be defined in ble_gap.h, while a GATTS parameter structure (such as ble_gatts_hvx_t) will be defined in ble_gatts.h.

In reality, the header files are nothing more than a list of which API function corresponds to which SVC operand, and which parameters they expect. However, when used from an application perspective, they look like normal C functions.

All functions in the API are non-blocking, but there are both synchronous and asynchronous functions. Most are synchronous, and return a result immediately but some start an operation that will result in the SoftDevice sending an event to the application at a later time. The nRF51 SDK help documentation contains a complete reference for all API functions, including their parameters and return values as well as message sequence charts showing which events are triggered by which functions.

The SoftDevice requires exclusive access to some peripherals and restricts access to others. For the ones that have restricted access, separate API functions are provided so that a peripheral can be safely shared between the application and the SoftDevice. All chip resources the SoftDevice needs to run are acquired when it is enabled, as described in the *S110 nRF51822 SoftDevice Specification*. For more information on the SoftDevice architecture, see ***Appendix A*** in the *nRF51 Reference Manual*.

3 Handling events

The SoftDevice is designed to be used by an event-driven application. Information from the SoftDevice to the application is delivered as an event through a software interrupt.

To start using the SoftDevice, the application initializes it with a clock source and a pointer to an error handler. To receive events, the application also has to enable the correct software interrupt. Usually an application will perform a use case specific initialization after enabling the software interrupt (for example setting up services, starting sensor readings), then start advertising and waiting for events (for example, a connection from a Central device).

To reduce power consumption, an application will typically put the chip to sleep while waiting for events. An event wakes up the chip and triggers the software interrupt and its handler. This allows the application to call an API function to retrieve the event.

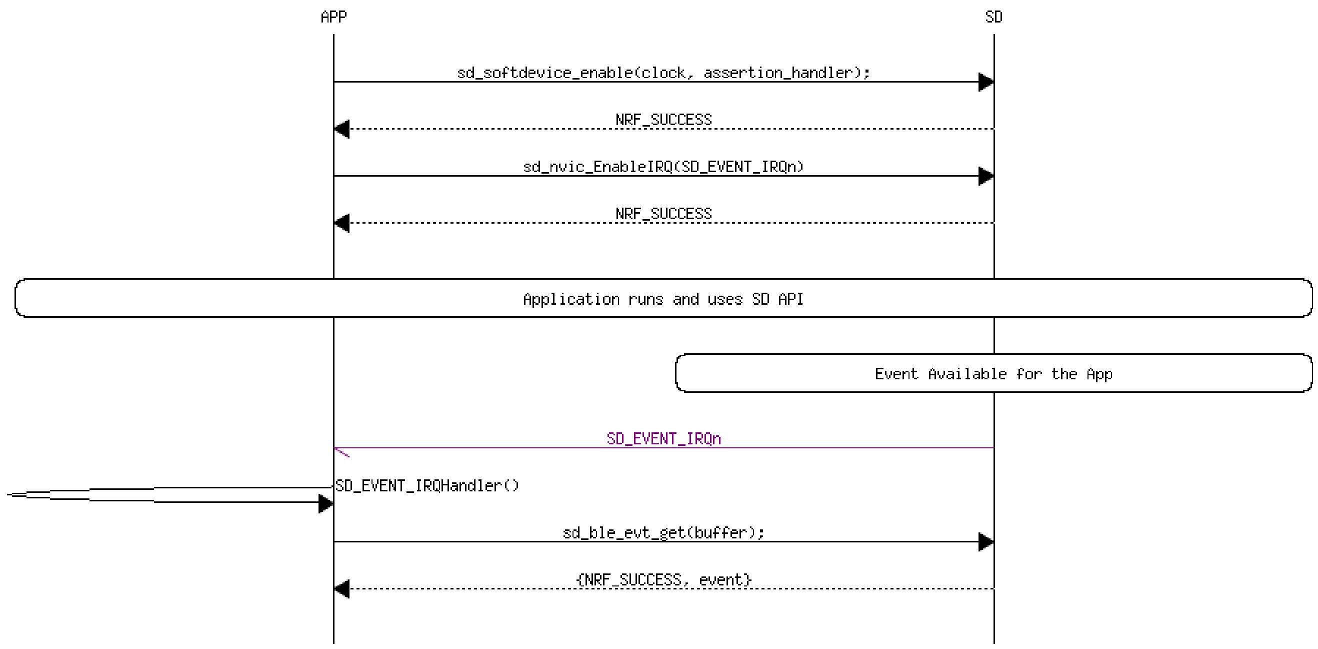
APP
SD
sd_softdevice_enable(clock, assertion_handler);
NRF_SUCCESS
sd_nvic_EnableIRQ(SD_EVENT_IRQn)
NRF_SUCCESS
Application runs and uses SD API
Event Available for the App
SD_EVENT_IRQn
SD_EVENT_IRQHandler()
sd_ble_evt_get(buffer);
{NRF_SUCCESS, event}

Figure 2 *Sequence chart showing the initialization and event handling of an S110 application*

Events are delivered to the application as a pointer to a C structure. This structure will contain different data depending on the type of event that the application can process and act upon. The event structures for different modules are defined in the header file of the corresponding layer (such as ble_gatts.h, ble_gap.h).

The nRF51 SDK provides a module called ble_stack_handler which makes initialization and event retrieval easier. In the SDK examples, all events are delivered to the application through this module to a function called ble_evt_dispatch. Because all events come through ble_evt_dispatch, it is used to send events to modules in the system. Each module then implements its own handler, managing only the events that the module needs.

Please see the SDK's ble_stack_handler module for details on how the interrupt handler is used to pass events on the application's event handler function (called ble_evt_dispatch in all SDK examples).

4 Buffer management

The SoftDevice has an internal buffer for data packets. Some on-air packets and API functions consume parts of the buffer, while others do not use it. Functions that use the buffer are clearly marked in their header file. A BLE_EVT_TX_COMPLETE event is given to the application each time a packet consuming a space in the buffer is sent on-air.

The application can choose between two different strategies for managing the buffer:

- **Utilizing a counter for the number of free spaces** – An application can decrement a counter when buffers are consumed and increment it when BLE_EVT_TX_COMPLETE events are received. There is an API method giving the initial number of buffers available, which must be used to initialize the counter. By keeping track of how much of the buffer is used, the application can know beforehand if a particular operation needing space in the buffer will succeed.
- **Handling return codes** – If there is no space left in the buffer, methods needing buffer space will give back a BLE_ERROR_NO_TX_BUFFERS return code. When this happens, whatever operation failed must be retried when a BLE_EVT_TX_COMPLETE event is received.

5 *Bluetooth* Services and Profiles

Because the SoftDevice generates events, all service implementations in the SDK are built to be event driven and have an initialization method and an event handler. In addition, most services have one or more API functions that trigger service specific actions. It is application specific when or how to call these methods. An example of this can be seen in the heart rate service, where the ble_hrs_heart_rate_measurement_send() function triggers the sending of a new heart rate measurement to the Central.

The initialization method of all service implementations takes two parameters; an instance structure and an initialization parameter structure. The instance structure contains persistent information for a service, including its state, handles to its characteristics, and similar. In this way, the structure functions as a handle to a specific instance of a service which all API functions need as their first parameter. The initialization parameter structure holds initialization values and other parameters that select the functionality of the service. It is only useful when setting up the service and can be removed immediately after initialization is finished. This structure can be seen in the services_init() of main.c in most of the SDK examples such as ble_app_hrs.

Since a profile in *Bluetooth* low energy is a use case description for a set of services, the profile implementations themselves are not visible in code. A profile is supported by having the correct services and implementing the behavior as defined in profile specifications released by the *Bluetooth* SIG. For example, the proximity application in the SDK (for both the evaluation board and nRFGo Motherboard) implements the Reporter role of the proximity profile by including Link Loss Service, Immediate Alert Service, and TX Power Service.

6 Helper modules

The nRF51 SDK contains several helper modules to make it easier to build nicely structured, event driven applications. There are modules to handle timers, buttons, UART communication, and so on. To the SoftDevice, all of these are event driven, and similar in that they need initialization and will give callbacks or events when things happen. These modules are documented in the nRF51 SDK help documentation.

7 Error handling

Inside the SoftDevice there are asserts that will fail if critical conditions are not true. During normal flow of an application which uses the API as documented, these will not fail. However, if there is a problem with program flow or API usage, it is reported to the application by calling the assert handler that was passed when initializing the SoftDevice. In this handler, a file name (from the SoftDevice source code) and a line number can be read out. In production code, the only way to recover from a SoftDevice assert is to reset the chip. If you ever see such an assert, please report it to technical support by creating a support case.

The SDK provides functionality useful for other error handling making it possible to easily check the error codes that the SoftDevice API functions return in a consistent way. If any method fails, a common error handler is called which can be used to track down the error. The assert handling methods delivered in the nRF51 SDK should not be used in a final product. They are intended to help with development but are provided as examples only.

Note: The error handler in SDK v 4.1.0 and v4.2.0 performs a system reset per default, which is not suitable for application debugging. The error handler should be either modified for debugging or a break point set prior to the reset when developing application code. For development, it is best to remove the error handler.

Liability disclaimer

Life support applications

Contact details

For your nearest distributor, please visit http://www.nordicsemi.com.
Information regarding product updates, downloads, and technical support can be accessed through your My Page account on our homepage.

Main office:	Otto Nielsens veg 12	Mailing address:	Nordic Semiconductor
	7052 Trondheim		P.O. Box 2336
	Norway		7004 Trondheim
	Phone: +47 72 89 89 00		Norway
	Fax: +47 72 89 89 89		

DNV

NORWEGIAN ACCREDITATION
No. S03

NS-EN ISO 9001 CERTIFICATED FIRM

Revision History

Date	Version	Description
June 2013	1.0	