UG10399

Embedded SW driver for ABS controller SDK User Guide

Rev. 2.1 — 10 June 2026 **User guide**

Document information

Information	Content
Keywords	Embedded, SDK, controller
Abstract	This document provides detailed instructions for installing, configuring, and using the embedded SW driver for ABS controller SDK.

1 Introduction

This document provides detailed instructions for installing, configuring, and using the embedded SW driver for ABS controller SDK.

This driver supports and provides high-level software solution for these analog parts:

- MC33SB0400: two-wheel antilock braking (ABS) controller for motorcycles
- MC33SB0401: one-wheel antilock braking (ABS) controller for scooter or moped

The FRDM-A-MOTABS2 freedom boards are evaluation platforms based on these chips. Refer to the related user guides and data sheets for detailed information.

2 MCU compatibility

The chapter describes the requirements for MCU peripherals and resources needed to control the analog device. The chapter briefly introduces the supported models. The chapter describes compatibility of board solutions based on the analog parts with selected MCUs.

2.1 Peripheral requirements

This section lists requirements on MCU peripherals and resources, which are critical for MCU capability to handle given parts.

- SPI module is required for communication (SI, SO, SCLK, CSB)
- GPIO is required for device reset (RSTB)

Note: *TPM/FTM is optionally required to generate PWM (PDI) or process wheel speed signal (WSOx, WSAI). Each of this advanced use cases is presented in a dedicated example.*

2.2 Supported ABS controller models

The section describes the supported models for the driver and explains the main differences and capabilities of the listed models.

MC33SB0400

- Four PWMed valve drivers up to 5.0 A (5.0 kHz)
- High-side predriver for valve protection
- Two-wheel speed sensor interfaces (active)
- Dual vehicle speed outputs
- Pump motor predriver up to 500 Hz PWM
- 16-bit SPI interface with watchdog
- K-line interface
- Warning lamp driver
- Die temperature warning
- Supervision

MC33SB0401 (derivate of SB0400)

- Two PWMed valve drivers up to 5.0 A (5.0 kHz)

2.3 Supported MCUs

The target MCU is S32K144 and selected subsets of MCUs from the Kinetis family. FRDM-KEAZ128 is a feasible candidate because it is automotive qualified.

Note: *The SW driver is built on AML (analog middleware layer), which creates an API abstraction layer for the SDK in use. The approach extends support for different layers similar to the SDK without changing the SW driver.*

Table 1 describes pin compatibility between ABS controller freedom boards and the selected MCUs. For Kinetis MCUs, using WSOx inputs and the virtual serial port simultaneously is not possible because both connect to the same MCU header pin.

Table 1. Pin compatibility of ABS controller freedom boards with selected MCUs

Pin Function	S32K144EVB	FRDM-KEAZ128	FRDM-KL25Z	FRDM-KL27Z	FRDM-KL43Z
RSTB	PTB1	PTF5	PTB3	PTE21	PTB3
MISO	PTB3	PTB4	PTD3	PTC7	PTD7
MOSI	PTB4	PTB3	PTD2	PTC6	PTD6
CSB	PTB5	PTB5	PTD0	PTC4	PTD4
SCLK	PTB2	PTB2	PTD1	PTC5	PTD5
PDI	PTD13	PTC5	PTA13	PTE31	PTA13
WSO1	PTA2	PTB0	PTA1 (UART1_TX)		
WSO2	PTA3	PTB1	PTA2 (UART1_RX)		
WSAI	PTB9	PTH1	PTA12	PTE25	PTA12
VSO_IN	PTD14	PTH2 (con. to MISO)	PTD5	PTA5	PTD2
RxK	PTB1	PTG5	PTA5	PTE24	PTA5
TxK	PTC10	PTG6	PTC8	PTC9	PTE29

Examples are given for **S32K144EVB** and **FRDM-KEAZ128**. See Section 4.2. For the rest of the MCUs, create a project based on an existing project and change the pin selection accordingly.

2.4 Freedom board settings

The jumper block on the FRDM-A-MOTABS2 provides a means of configuring the boards for use with additional MCUs. Jumper J16 (DOSV) allows you to select between 3.3 V and 5.0 V, depending on the requirement of the MCU being used. Ensure that you set jumper J16 to the proper voltage level (for S32K144 and KEAZ128, set DOSV to 5 V).

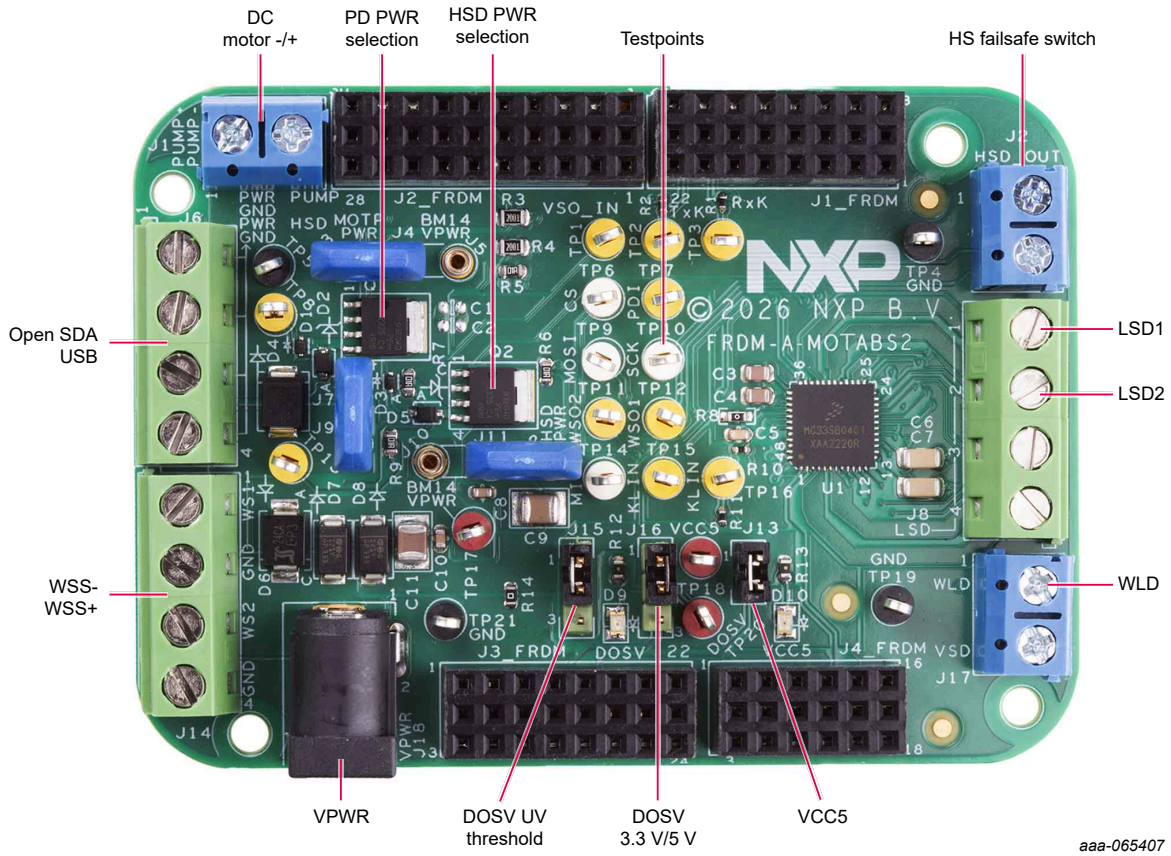


Figure 1. FRDM-A-MOTABS2 settings

3 Embedded SDK SW driver for ABS controller

This chapter explains the functionality, settings, and usage of the driver, including configuration and functions. This text is intended to summarize the capabilities of the driver.

The programmer's guide is generated API documentation for the driver. It provides more detailed information about configuration structures and functions with parameters. Refer to the code and related commentary to gain better insight into the driver.

3.1 Driver user configuration

The driver is static and uses a single entry point defined by its user configuration structure. The user fills the structure with values according to the desired device setup. The initialization function receives this configuration directly.

Configuration options (depends on the selected model):

1. **Clock settings and monitoring** (internal clock monitoring function, oscillator clock frequency, and modulation band).
2. **Low-side drivers for inductive load** (high-side fail-safe switch, output PWM frequency, sink current for open load detection).
3. **Pump motor predriver** (control – SPI/external, overcurrent masking/filter time, slew rate, load dump).
4. **Wheel speed sensor interfaces** (sensor types, I/O signal muxing, OC filter time, digital filter time, internal counter, leakage current tracking).
5. **Auxiliary drivers** (warning lamp functionality – enabled/disabled).
6. **Watchdog** (LFSR seed).

For a more detailed description of the user configuration structure, refer to the programmer’s guide.

3.2 Driver API

The SW driver provides an API for dynamic real-time device configuration in user code. Table 2 summarizes the available functions.

Table 2. ABS controller SW driver API

Function	Description
Get Default User Config	Fills user configuration structure with default values. Note that all of the present drivers are turned off by default, except the HS fail-safe switch.
Get Register Config	Initializes register array based on user configuration. This register array can be directly written to the device registers.
Init	Initializes the driver based on user configuration.
Reset Device	Resets the device. The init function must be called before further usage of the driver. After reset, the device registers go to their default states and must be reinitialized.
Get Reset Pin Val	Gets level of the reset pin, which also acts as a fault pin.
Read Register	Reads data from the selected register.
Write Register	Writes data to the selected register.
Update Register	Updates the content of the selected register. Only bits given by the mask are affected.
Feed Watchdog	Sends computed monitoring result to the ABS controller.

Table 2. ABS controller SW driver API*...continued*

Function	Description
Set LFSR State	Stores received LFSR state for the next computation.
GetStatus	Reads register related to the selected status request. The user can use enumeration to access items of the array.
Clear Faults	Clears all faults related to the drivers of the ABS controller.
Set Driver State	Controls state of the ABS controller drivers (LSD, PD, WLD, WSS). The user can use enumeration to access items of the array.
Get Measurement	Reads selected ADC value internally measured by the ABS controller. Result is converted to the expected units.
LSD PWM-specific functions	
Set LsdPwm Duty	Sets target PWM duty for selected LSD.
Get LsdPwm Duty	Gets actual PWM duty for selected LSD.
WSS-specific functions	
Set Speed Cntr Src	Muxes WS_s to select WSOx as the source signal for the internal speed counter.
EnDisSpeedCntr	Enables or disables the internal wheel speed counter.
ResetSpeedCntr	Resets internal wheel speed counter.
GetSpeedCntrVal	Gets internal wheel speed counter value.
MuxWSAI	Muxes WSAI_s to select WSI10_x as the source signal for WSAI.
MuxVSO	Muxes VSO_sel and VSO_s to select source signal for VSO. Either VSO_IN or WSOx can be routed to VSO.
EnDisLeakTrack	Enables or disables leakage current tracking for wheel speed sensor interfaces.
GetWssStatus	Gets status information related to the selected wheel speed sensor interface.
GetWssDataType2	Gets additional data from the wheel speed sensor of type 2.
GetWssDataType3	Gets additional data from the wheel speed sensor of type 3.
GetIsoKStatus	Gets status information related to the ISOK interface.

For a more detailed description of the SW driver API (function signatures, parameters) refer to the programmer’s guide.

3.3 Utilized low-level drivers

The SW driver uses low-level MCU peripheral drivers for communication, control, and related functions. The SW driver functionality depends on these low-level drivers.

The latest SW driver is built with the NXP SDK using GCC-6.3. The SW driver remains compatible with an older SW driver built on AML using GCC-4.9. To support portability across different SDKs, the AML layer unifies the API for selected low-level drivers.

Reusing existing AML-based low-level drivers accelerates development and provides verified solutions for common tasks.

Figure 2 illustrates the SW driver architecture for communication and control.

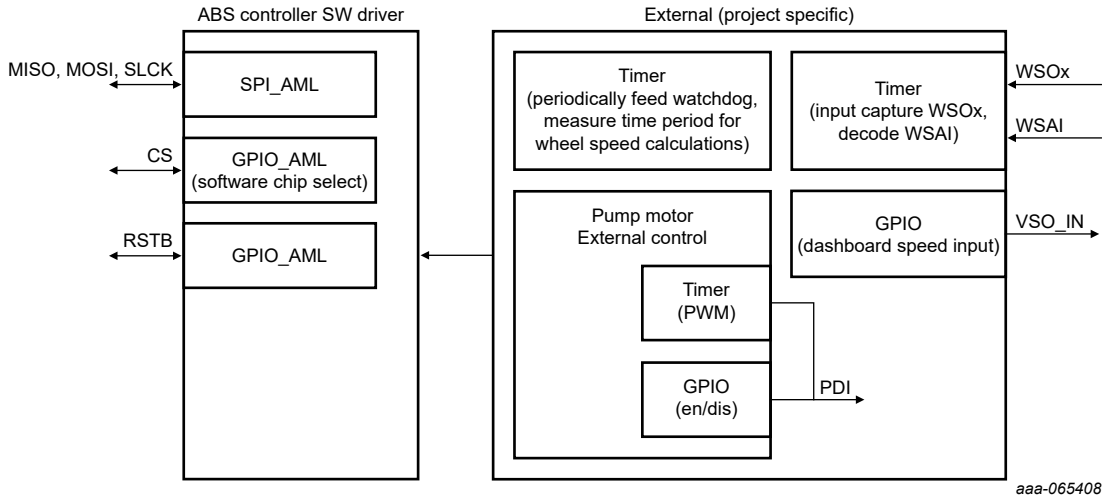


Figure 2. SW driver architecture

3.4 Required driver setup

The user must follow these requirements to use the SW driver correctly:

1. Specify used SDK in define **SDK_VERSION** in file **common.h**.
2. Specify used ABS controller model in define **ABS_MODEL** in file **abs/abs_model.h**. This action adjusts the SW driver according to the specified model needs and available features.
3. Customer should include **s32_core_cm4.h** in **devassert.h** file manually, otherwise it may report some error.

3.5 Implementation notes

This section describes how the SW driver handles specific features of the ABS controller devices.

3.5.1 Fault detection and handling

The SW driver provides functions that allow application code to read device status information and react to faults. Table 3 lists the functions used for fault detection and handling.

Table 3. Functions related to fault handling

Function	Description
ABS_Init()	Recovers from a fault that caused a register's reset and set the reset pin to low.
ABS_ResetDevice()	Resets the device and its registers to its initial state. Temporarily changes the direction of the reset pin.
ABS_GetResetPinVal()	Gets reset pin value. Since the reset pin also acts as a fault pin, its level represents a possible device fault state (low).
ABS_GetStatus()	Reads status information provided by the supervision module. User can use register map for specific model to interpret the content of status registers in a **human way**.
ABS_ClearFaults()	Clears all faults related to drivers present on the ABS controller device.

3.5.2 Wheel speed sensor

This SW driver provides a basic set of functions covering actions done by SPI to control wheel speed sensor interfaces. This feature is present on MC33SB0400 and MC33SB0401 models. Table 4 and Figure 3 show the mapping of functions to the wheel speed sensor interface.

To use WSS, the user sets the required items in the user configuration structure. Section 3.1 describes the configuration details. The user application provides three options to calculate the final wheel speed value:

1. Calculate speed based on the internal counter value. Hardware performs all logic, and the user tracks internal counter changes over time using an MCU timer. Known wheel physical parameters allow calculation of the final speed.
2. Count pulses on the WSOx output using an MCU timer in input capture mode.
3. Decode the WSAI signal on the MCU side. This option serves as a safety feature.

Table 4. WSS-specific functions

Function	Description
SetSpeedCntrSrc	Muxes WS_s to select WSOx as the source signal for the internal speed counter
EnDisSpeedCntr	Enables or disables the internal wheel speed counter
ResetSpeedCntr	Resets internal wheel speed counter
GetSpeedCntrVal	Gets internal wheel speed counter value
MuxWSAI	Muxes WSAI_s to select WSI10_x as the source signal for WSAI
MuxVSO	Muxes VSO_sel and VSO_s to select source signal for VSO. Either VSO_IN or WSOx can be routed to VSO
EnDisLeakTrack	Enables or disables leakage current tracking for wheel speed sensor interfaces
GetWssStatus	Gets status information related to the selected wheel speed sensor interface
GetWssDataType2	Gets additional data from the wheel speed sensor of type 2

Table 4. WSS-specific functions*...continued*

Function	Description
GetWssDataType3	Gets additional data from the wheel speed sensor of type 3
GetIsoKStatus	Gets status information related to the ISOK interface

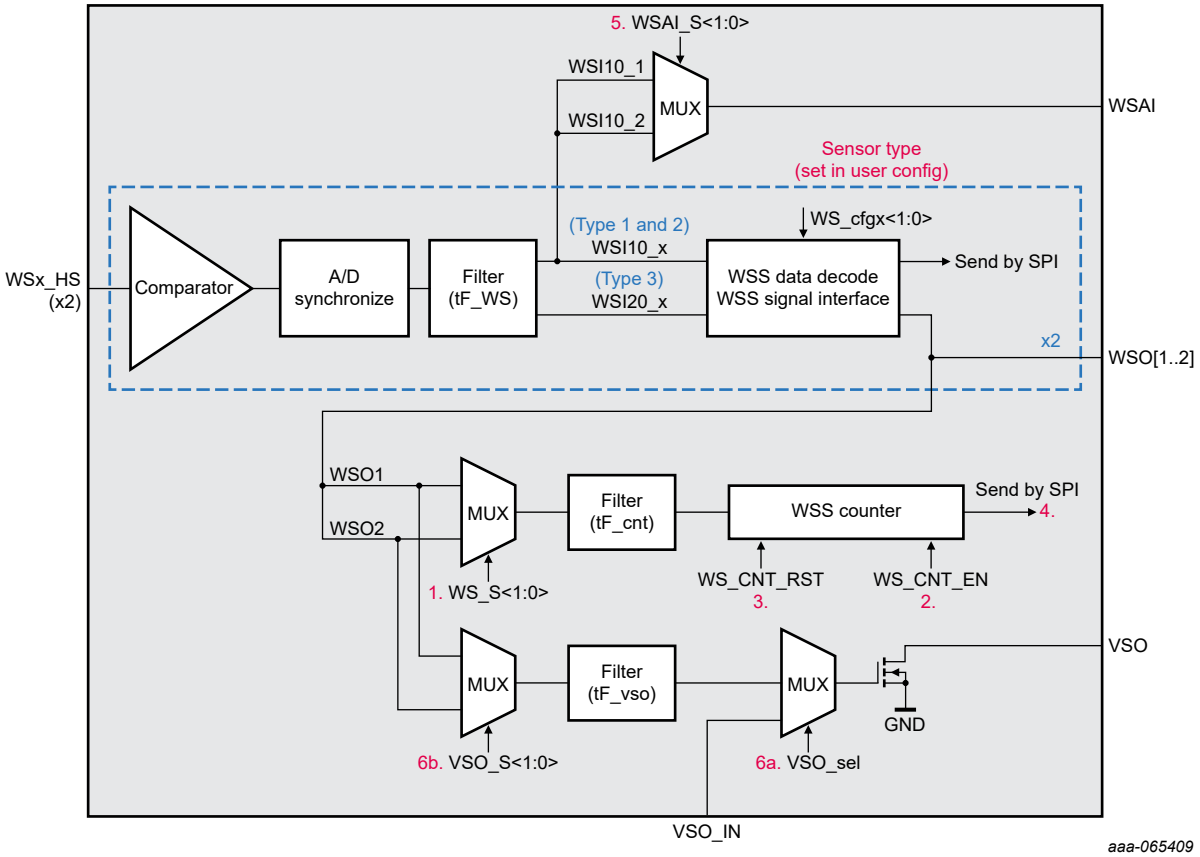


Figure 3. WSS functions mapping

For more detailed information on the wheel speed sensor, refer to MC34SB0400 and MC34SB0401 datasheets.

4 Install Software

S32 design studio is a SW driver built on an NXP SDK. This section describes how to install S32 design studio and use it for application development.

4.1 Installing S32 design studio

The procedure explains how to obtain and install the latest version of S32 Design Studio, Section 2.2 in the guide.

Note: *The driver and some package examples target S32 Design Studio 2.2. If the IDE is already installed on the system, skip the steps in this section.*

1. Obtain the latest S32 Design Studio installer from the NXP website.
2. Run the executable file and follow the onscreen instructions.

For S32 Design Studio, the SDK library is distributed with the IDE and does not require manual addition or linking.

4.2 Get the SW driver and example projects

This procedure explains how to obtain the latest version of SW driver and example projects.

1. Download zip file with driver and example projects from the link.
2. Unzip the downloaded file and check the folder contents listed in Table 5.

Table 5. Content of the downloaded zip file

Folder Name	Folder Contents
S32DS_Examples	Example project folder for S32 Design Studio 1.2 or newer.
ABS_S32K144_BM14	The example shows how to monitor the status of the ABS controller using the ABS controller SW driver. The example covers reading measurements and status information and performing actions such as clearing faults to ensure proper driver operation.
	The purpose of this example project is to show how to implement PWM for the Pump Motor Predriver. This project demonstrates two ways: 1. With SPI control − output state of pump motor predriver is driven by SPI commands. The advantage of this solution is that there is no required additional MCU pin for direct control of Pump Motor Predriver. 2. Generated by a timer − PWM signal is generated on the MCU side by the TPM peripheral. This solution consumes less CPU time.
	The example project demonstrates how to work with the wheel speed sensor interface. The example does not cover calculation of the final wheel speed, which requires additional hardware equipment such as a wheel. The example focuses on counting the number of edges within a given time period.
Programmer_Guide	This folder contains detailed API documentation for this SW driver.

4.3 Import an example project into S32 design studio

The following steps show how to import an example from the downloaded zip file into S32 Design Studio.

1. In the S32 Design Studio menu bar, click **File > Import**, select **General > Existing Projects into workspace** in the pop-up window, and select **Next**, as shown in Figure 4.

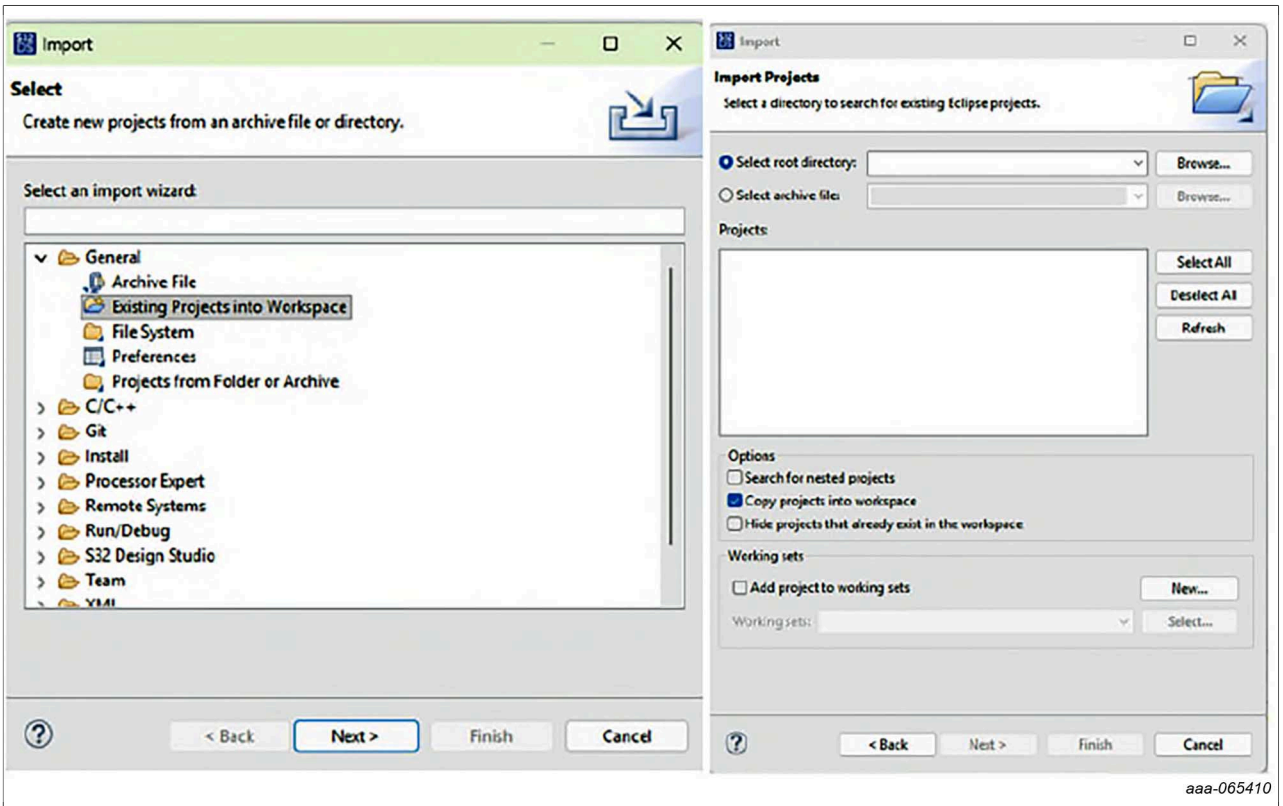


Figure 4. Importing an example project (a)

2. Click **Browse**, locate the folder containing the unzipped example files, find the **ABS_S32K144_BM14** folder, select a project to import as shown in Figure 5, and select **OK**.

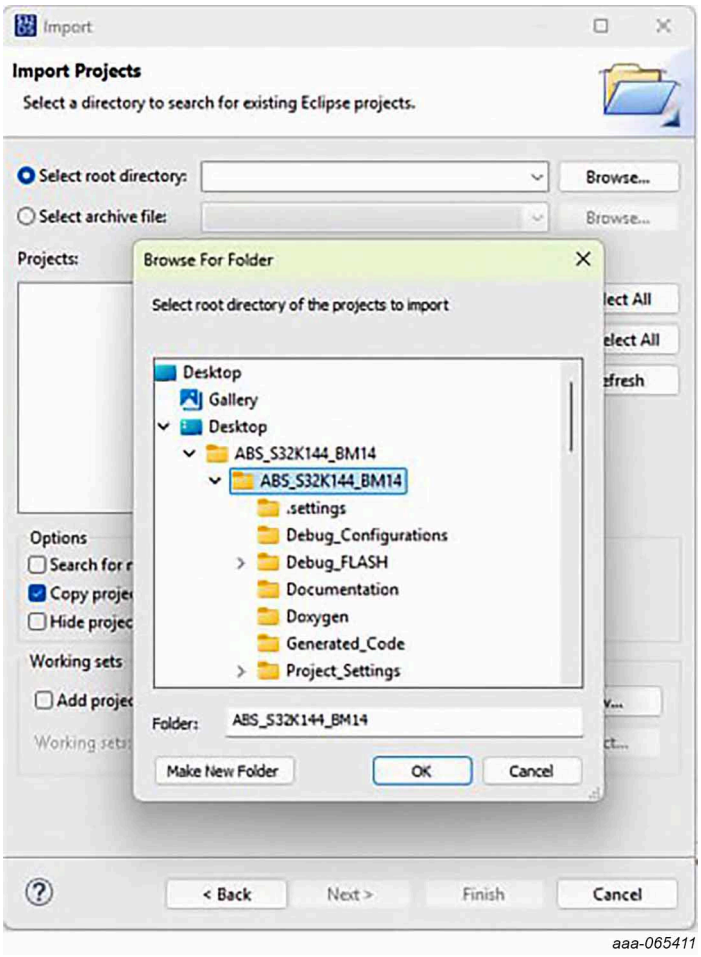


Figure 5. Importing an example project (b)

3. With the project loaded in the **Select root directory** box, select the **Copy projects into workspace** checkbox and select **Finish**. The project appears in the S32 Design Studio workspace, where you build and run it, as shown in Figure 6.

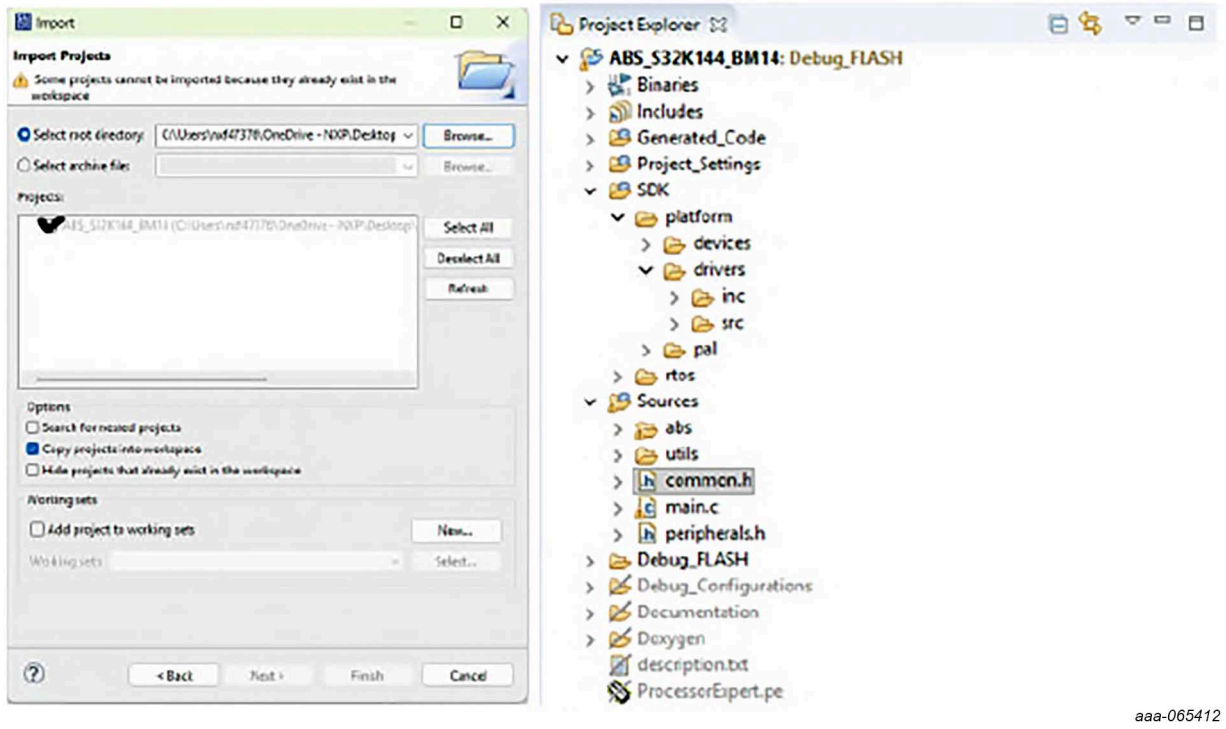


Figure 6. Importing an example project (c)

4.4 Create a new project with ABS controller SDK SW driver

If you do not use the example projects, the following instructions describe how to create and set up a new project that uses ABS controller SDK SW driver.

To create a project, follow the steps below:

1. In the S32 Design Studio menu bar, select **File** >**New** >**New S32DS Project**, as shown in Figure 7.

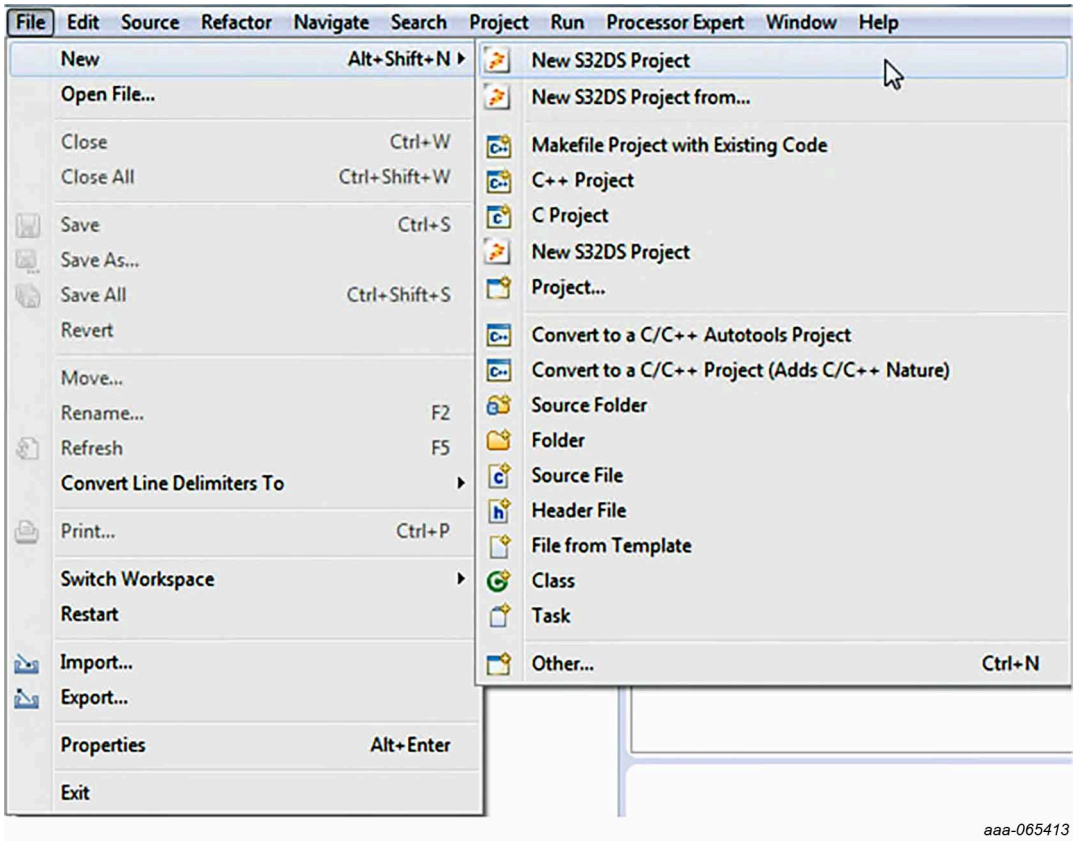


Figure 7. New S32DS Project Wizard

2. When the **New S32DS Project** box opens, enter a project name into the text box, select the MCU for your project (S32K144 used in this tutorial) and then select **Next**, as shown in Figure 8.

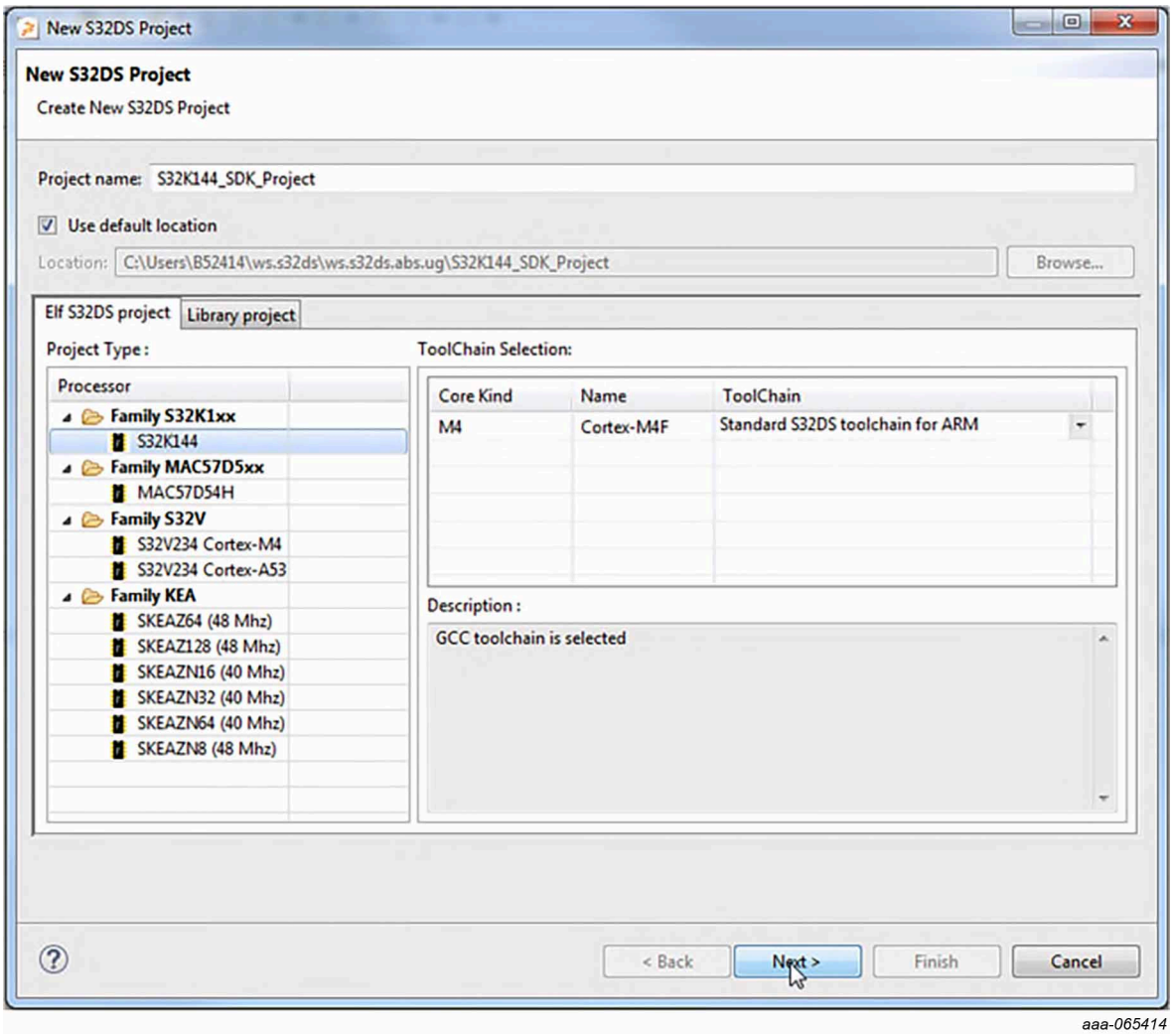


Figure 8. S32DS Project Name and MCU Board Selection

3. In the next step, set the required project settings such as the used SDK or frameworks, as shown in Figure 9, and then select **Finish**.

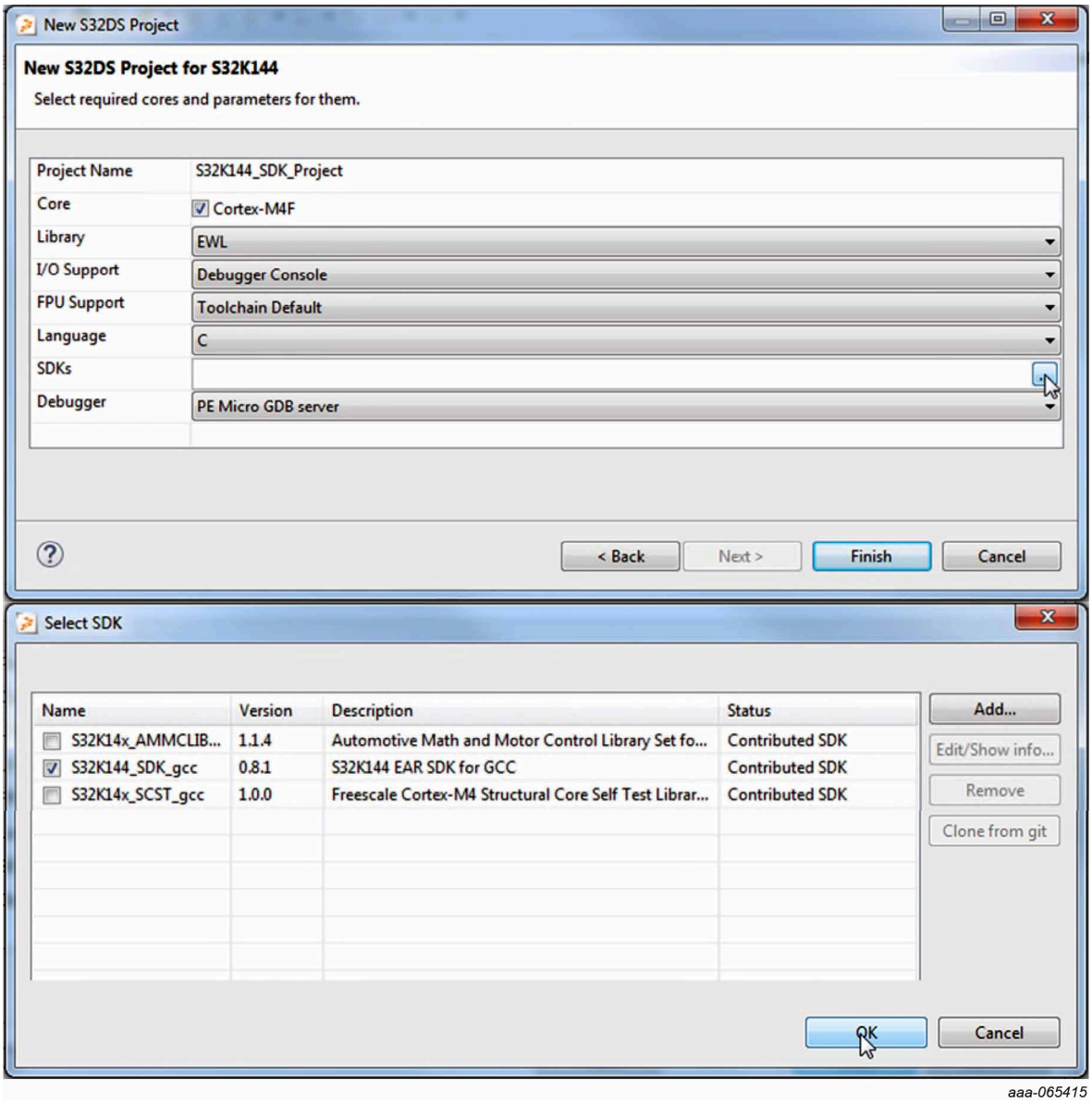


Figure 9. S32DS Project Settings and SDK selection

4. Figure 10 shows the Project Explorer panel and **main.c** content after creation of a new project.

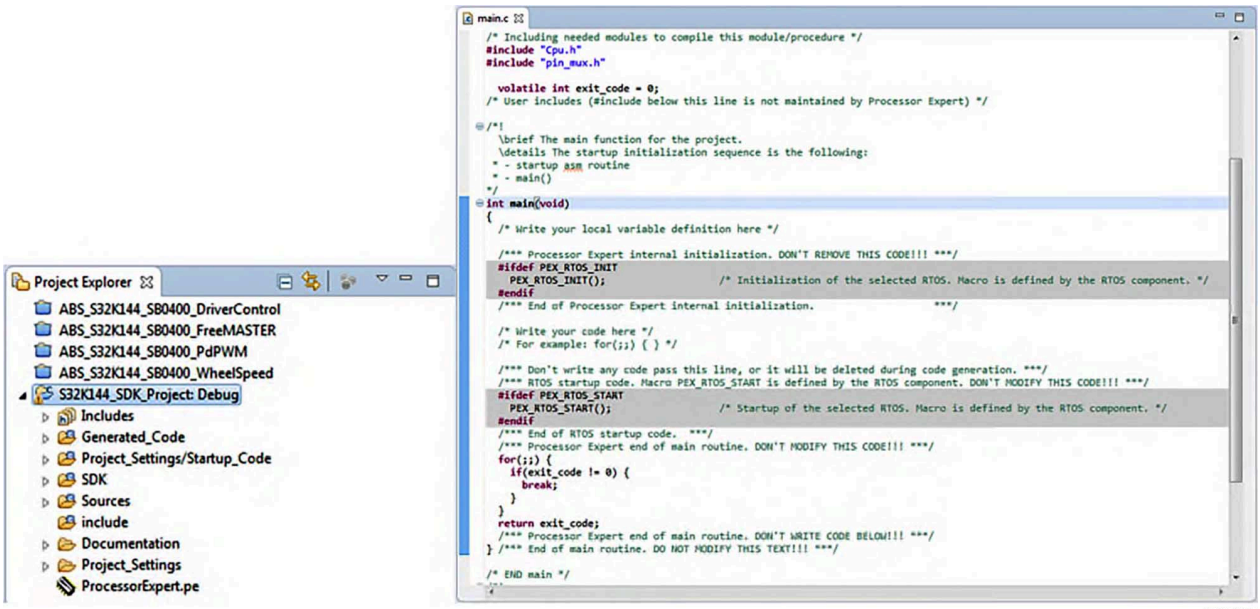


Figure 10. Created S32DS Project in S32DS Project Explorer and main.c template.

4.4.1 Setting up the processor expert components

This section describes how to configure an MCU using Processor Expert components, which facilitate the whole setup process by providing a user-friendly GUI for configuration of common parts such as clocks, pin muxing, and peripherals.

1. First, ensure the workspace is set to Processor Expert view by selecting **Processor Expert → Show Views** in the S32DS menu, as shown in Figure 11. This action adds tabs to the workspace, including Components, Component Inspector, and Component Library.

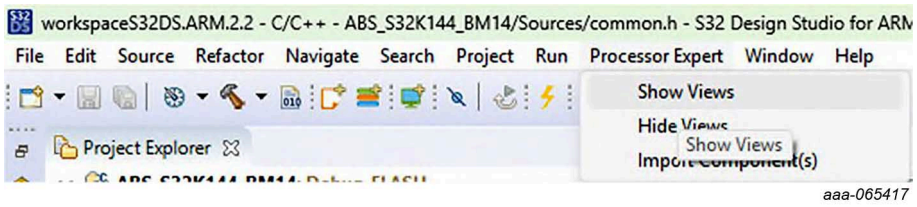


Figure 11. Processor expert show view

2. To run the SW driver on the selected MCU, configure the used SPI and GPIO pins. Select the correct pins in the PinMux component for the used MCU according to Section 2.3, as shown in Figure 12. Use one of the example projects as a template for the step.

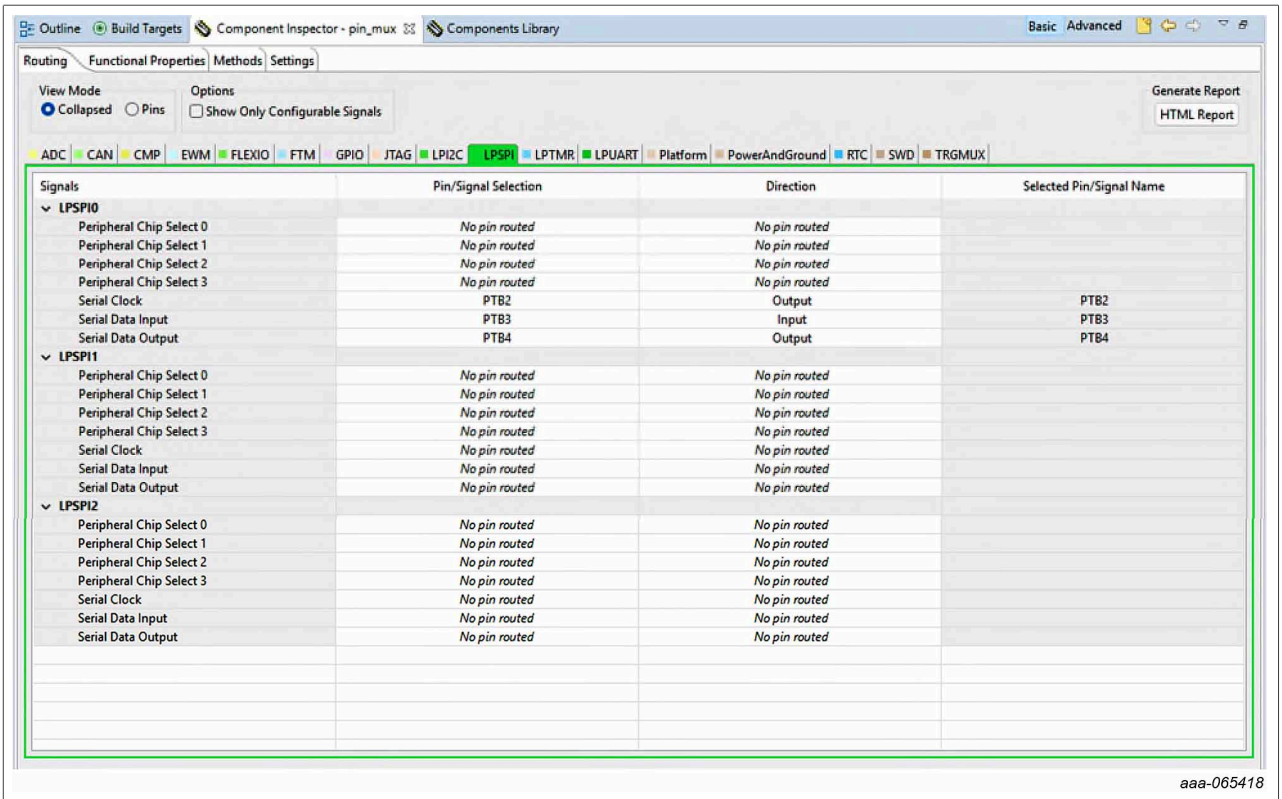


Figure 12. PinMux component in S32DS project

3. Add components for the required MCU peripherals from the S32 SDK repository, including LPSPI (spi_pal). If you use a debug console, add LPUART (lpuart) components, as shown in Figure 13.

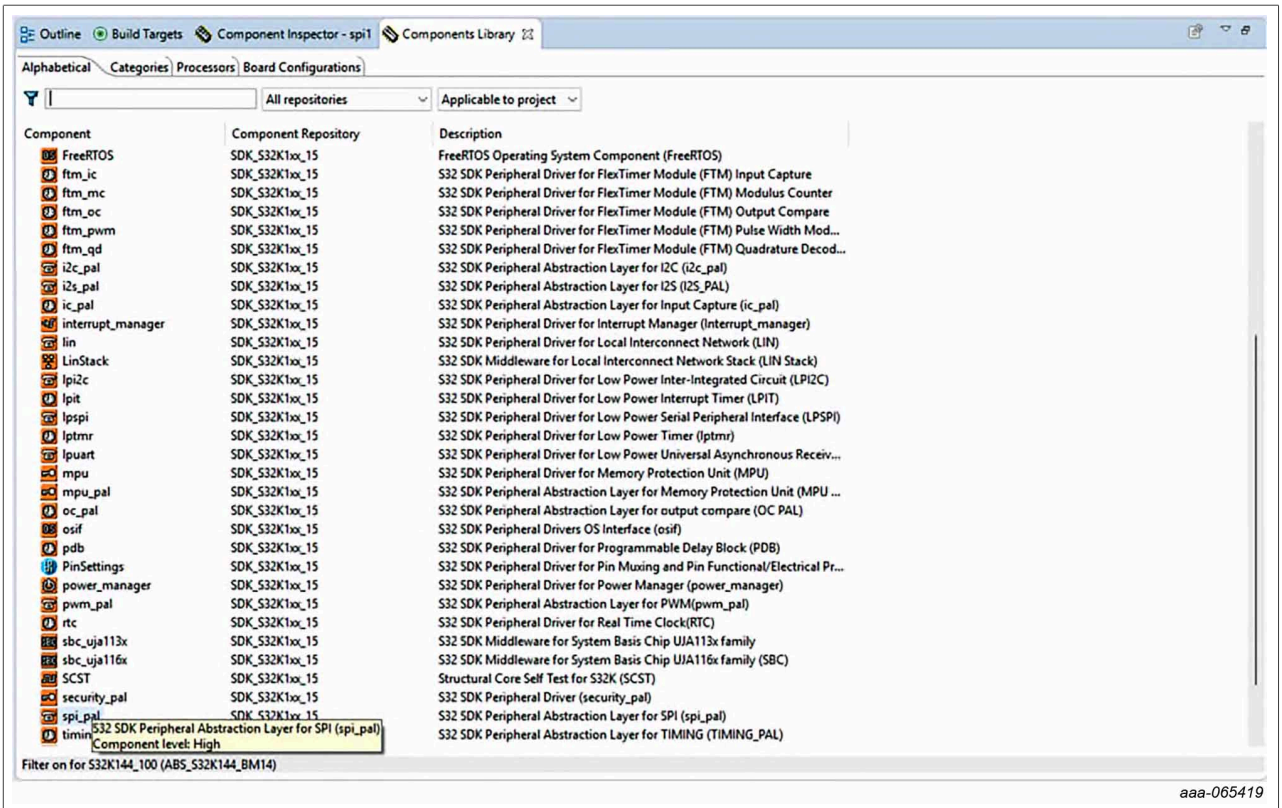


Figure 13. Add components from S32 SDK repository

4. All added components now appear in the Components tree, as shown in Figure 14. The S32 SDK and Processor Expert interface are closely connected. The project SDK folder includes only the peripheral drivers that correspond to the components used in the project. In S32DS, Processor Expert serves as a graphical interface for configuring S32 SDK driver structures. The code then passes these structures to initialization functions.
   Note: *You notice the difference between Processor Expert usage based on LDD and Processor Expert used as a GUI frontend for the S32 SDK. The former approach generates and modifies the entire SW driver code. The later approach generates only configuration structures. The approach improves safety by maintaining a static SW driver and a single entry point for initial configuration.*

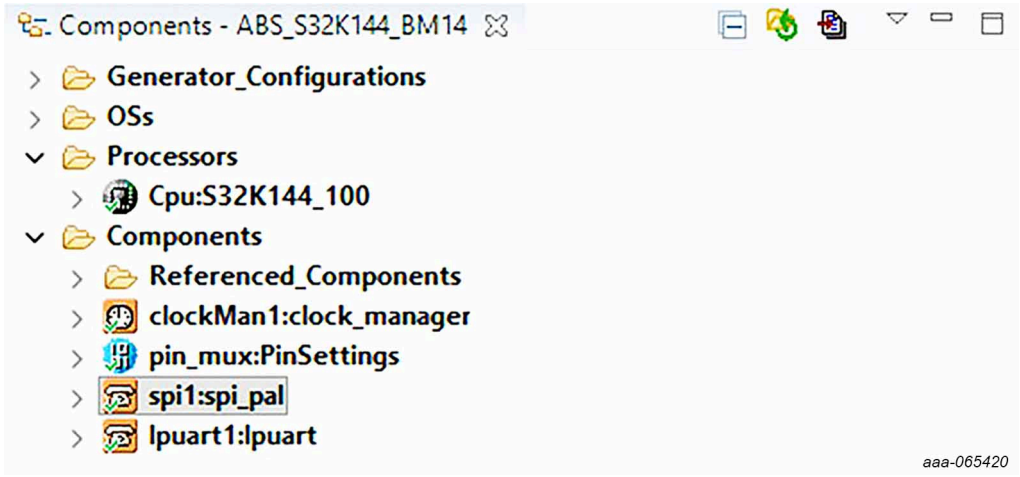


Figure 14. Utilized components in S32DS project

5. The next step configures the added components in the Processor Expert Component Inspector. A common set of configuration steps applies to all projects. If required options are unclear, refer to the datasheets of the used MCU and MC33SB0400/401 or to existing examples.
   a. Clock configuration should be done using clock_manager. There you can select clock sources, frequencies, prescalers and clock gates for the MCU and its peripheral.
   b. Perform pin muxing in Pin_Settings. The process is already explained in previous steps. For each MCU peripheral that uses I/O pins, select the proper routing (muxing).
   c. Configure peripheral components such as spi_pal, lpspi, and lpuart in dedicated components. See the configuration of the LPSPI component in Figure 15 as an example.

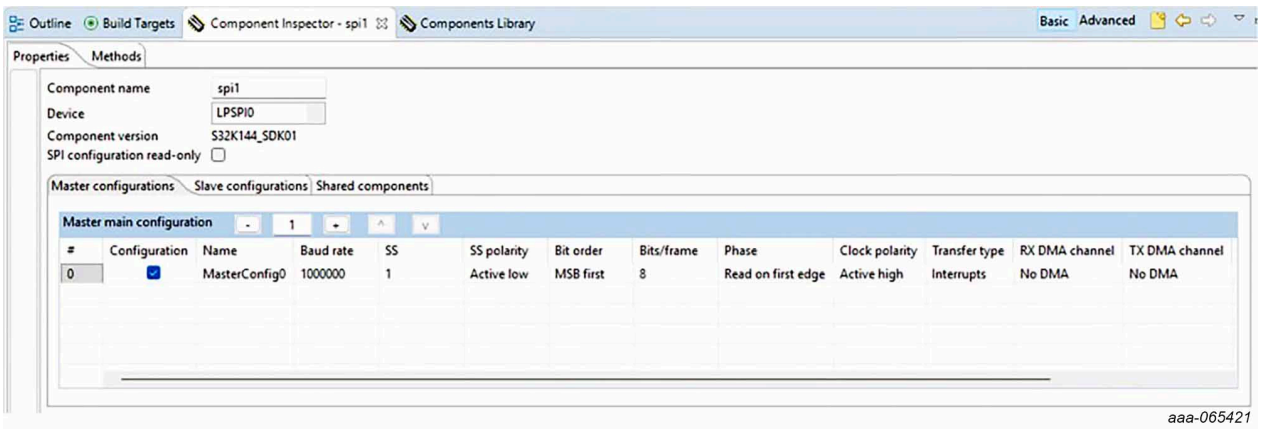


Figure 15. Configuration of spi_pal component using processor expert Inspector

6. The configuration of the MCU and the used peripherals is complete, and the next step generates code for the components using Processor Expert. Select the icon shown in Figure 16. The Generated_Code folder in the project folder contains the generated files. Later, inspect these files to identify the names of configuration structures passed to the SDK driver initialization functions.

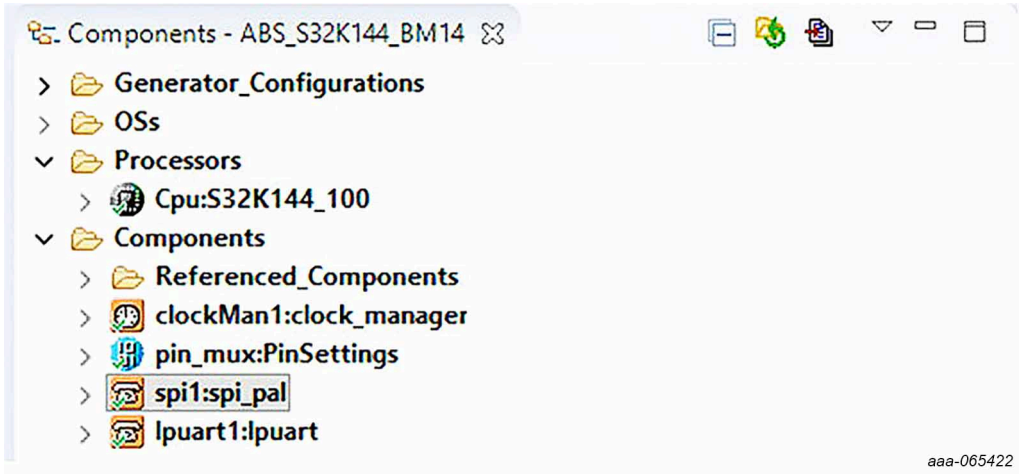


Figure 16. Generate processor expert code

4.4.2 Add the ABS SW Driver to the Project

This section describes how to add the ABS Controller SW driver to the project.

1. Copy the content of the **ABS_S32K144_BM14 driver** to the **Sources** folder in the newly created project. Figure 17 shows the list of copied files.

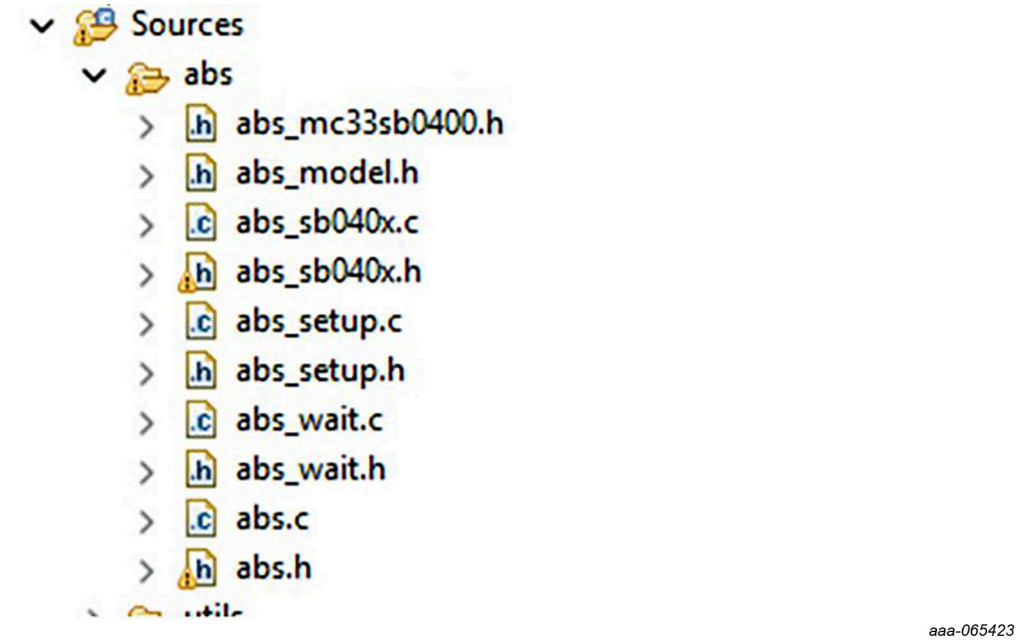


Figure 17. Add the ABS Controller SW Driver to the project

2. Add the line highlighted in Figure 18 to **main.c** to access the ABS controller SW driver in user code.

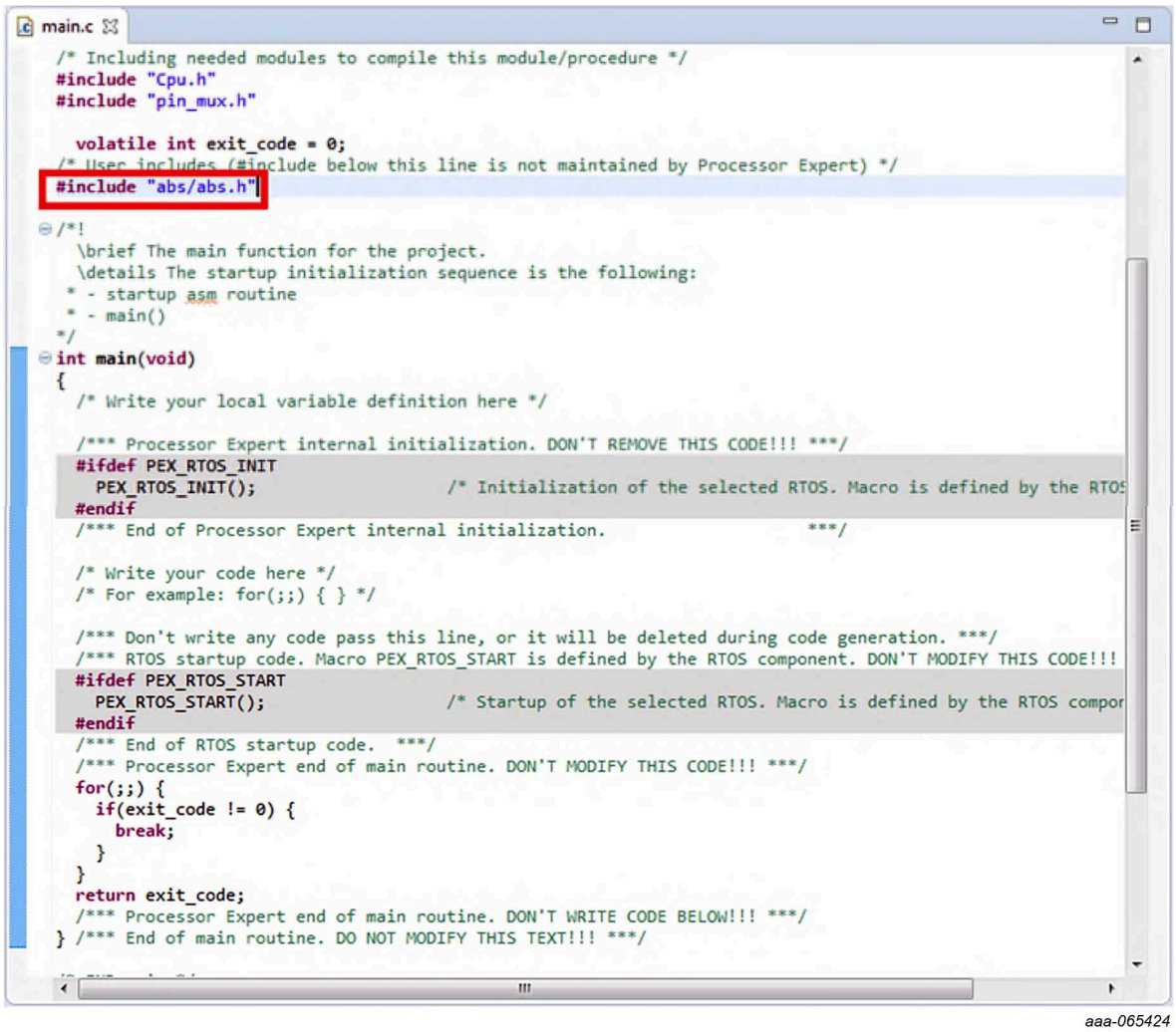


Figure 18. Link ABS Controller SW Driver

4.4.3 Setting up the ABS SW driver

After creating the new project and adding the ABS controller SW driver, set it up and associate it with the preconfigured MCU peripherals. Read Section 3, which describes driver capabilities and the required configuration steps.

1. Change used ABS controller model in **abs/abs_model.h** accordingly.
2. Create a variable of type **abs_drv_data_t** and pass it to all used functions. The variable stores configuration of MCU peripherals and internal data of the ABS controller SW driver. Placing the variable under user control ensures that the SW driver remains independent of the number of used instances. The number of instances is limited only by MCU resources. Ensure that the variable remains accessible during runtime. Declare the variable in the function **main()** or as a global variable.
3. Configure the ABS SW driver as follows, as shown in Figure 19. Apart from **ABS_GetDefaultUserConfig()**, the user changes individual items according to user requirements. The configuration structure maps internally to the device register array based on the selected model.
4. Associate the used SPI and GPIOs with ABS SW driver runtime data. The driver core requires access to GPIOs for reset and software chip select and to the SPI module for communication. Build extended

functionality such as PWM and WSS processing on the SW driver using additional peripherals, such as FTM, PIT, and GPIOs. Use GPIOs for other required pins, such as PDI, WSOx, VSO_IN, and WSAI.

5. Use the **ABS_SetupSPI()** function optionally to initialize the SPI peripheral with the configuration generated by the Processor Expert component named fsl_lpspi. In S32DS, the **ABS_SetupSPI()** function passes the configuration structure to the SDK SPI driver initialization function. The **ABS_SetupGPIOs()** function is not used in S32 SDK. Initialize pins using the **Pins_DRV_Init()** function with the configuration structure generated by the PinMux component.
6. Finally, call the ABS SW driver initialization function. The user must pass a reference to the configured MCU peripherals, which are part of the driver data structure, and include the user configuration of the device. The approach separates responsibilities between configuration of MCU peripherals and their usage.

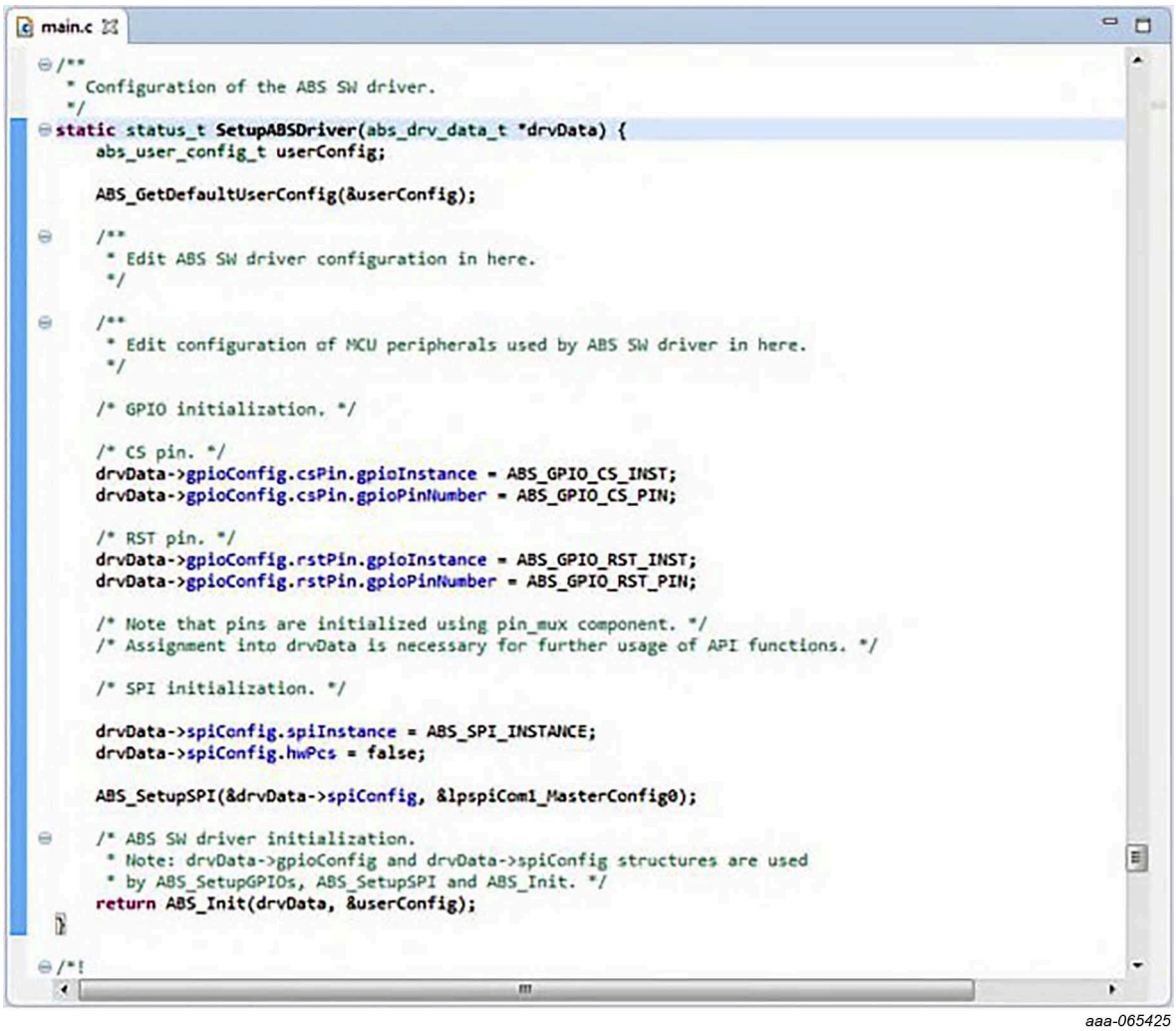


Figure 19. Setup function for ABS SW driver

4.4.4 Writing your application code

All application code resides in the **Sources** folder in the project directory. You can modify the code in **main.c** while retaining the original usage directions -- related comments.

Follow the steps below to get the code working with the ABS controller:

1. Call initialization functions for all used MCU peripherals, except SPI when ABS_SetupSPI() is used earlier. Pass pointers to the generated configuration structures found in **.h** files in the **Generated_Code** folder. Figure 20 shows an example of the initialization. The code reuses all previously described steps and assumes that the user has configured clocks, pin muxing, peripherals, generated code, and implemented the ABS SW driver setup function. Refer to the code commentary to understand the implementation.

main.c

```
/*!
  \brief The main function for the project.
  \details The startup initialization sequence is the following:
 * - startup asm routine
 * - main()
 */
int main(void)
{
  /* Write your local variable definition here */
  status_t status;                    /* Status variable. */
  abs_drv_data_t absData;            /* ABS SW driver data needed by the
                                       driver's functions. */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  #ifdef PEX_RTOS_INIT
    PEX_RTOS_INIT();                  /* Initialization of the selected RTOS. Macro is defined by the RTOS component. */
  #endif
  /*** End of Processor Expert internal initialization.                    ***/

  /* Initialize and configure clocks
   *     -   see clock manager component for details
   */
  CLOCK_SYS_Init(g_clockManConfigsArr, FSL_CLOCK_MANAGER_CONFIG_CNT,
          g_clockManCallbacksArr, FSL_CLOCK_MANAGER_CALLBACK_CNT);
  CLOCK_SYS_UpdateConfiguration(0U, CLOCK_MANAGER_POLICY_FORCIBLE);

  /* Initialize pins
   *    -   See PinSettings component for more info
   */
  Pins_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);

  /* Initialize debug console (initialize LPUART based on PEx configuration). */
  status = DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_TYPE, \
          &lpuart1_State, &lpuart1_InitConfig0);
  if (status != kStatus_Success)
  {
      /* Console is not initialized... */
  }
```

aaa-065426

Figure 20. Initialization of MCU and used peripherals

2. Call the setup function for the ABS SW driver (in this tutorial called SetupABSDriver()), see Figure 21.
3. After the SW driver configuration is complete, use the available functions to build the application. Refer to the Programmer's Guide for function signatures and required parameters. Also review the **sources/abs/abs.h** header file, which contains prototypes for all available functions.
4. Call **ABS_FeedWatchdog()** approximately every 50 ms to keep the ABS controller operating. You can use either periodic interrupts or an infinite loop with nested delays. The delay prevents excessive SPI communication overhead and avoids overloading the MCU.

main.c

```
PRINTF("S32 SDK ABS SW driver.\r\n");

/* Initialize ABS driver. */
status = SetupABSDriver(&absData);
if (status != kStatus_Success)
{
      PRINTF("SetupABSDriver failed with error code %d.\r\n", status);
}
else
{
    PRINTF("\r\n-----------------------------------\r\n");
    PRINTF("This example project monitors status of the ABS driver.\r\n");
    PRINTF("PD, WLD, HD, WSS1/2 and LSD1 drivers are are continuously monitored.\r\n");

    /* Watchdog must be reset periodically. */
    status = ABS_FeedWatchdog(&absData);
    if (status != kStatus_Success)
    {
        PRINTF("FeedWatchdog failed with status code %d.\r\n", status);
    }

    /* Turn on high-side fail safe switch. */
    status = ABS_SetDriverState(&absData, absDrvHighSideFailSafe, true);
    if (status != kStatus_Success)
    {
        PRINTF("SetDriverState failed with status code %d.\r\n", status);
    }

    /* Start application. */
    if ((status = StartApp(&absData)) != kStatus_Success)
    {
        PRINTF("StartApp failed with status code %d.\r\n", status);
    }

    /* Feed watchdog. */
    while (1)
    {
        status = ABS_FeedWatchdog(&absData);
        if (status != kStatus_Success)
        {
            PRINTF("FeedWatchdog failed with status code %d.\r\n", status);
        }

        WaitMS(WATCHDOG_DLY_MS);
    }
}
```

aaa-065427

Figure 21. User application using ABS SW driver

Writing the user application follows the same approach as shown in the previous steps. For more complex ABS controller use cases, refer to the included example projects.

Note: *The tutorial and all included examples do not cover configuration and initialization of the SBC on the S32K144 freedom board. The scope of the SW release excludes this functionality. For SBC configuration and initialization, refer to the documentation related to the SBC used.*

4.4.5 Compiling, downloading, and debugging

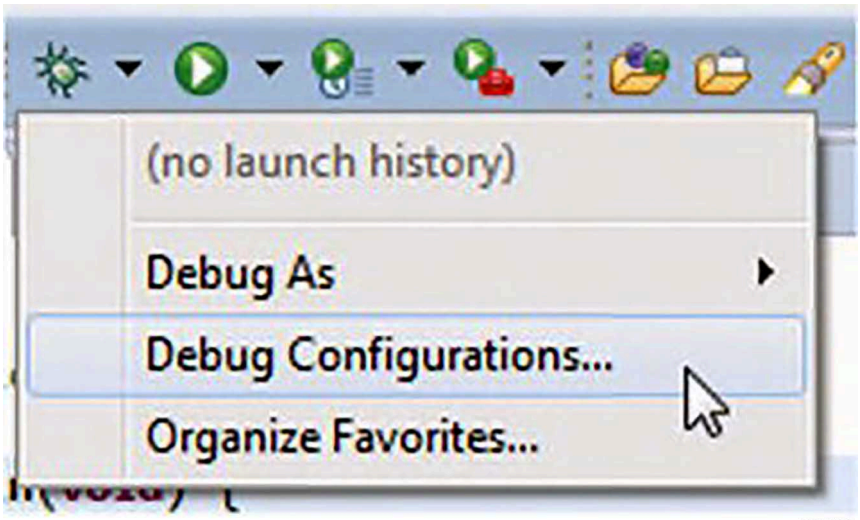
To compile a project, select the compile icon in the toolbar, as shown in Figure 22.

aaa-065428

Figure 22. Compiling the project

The process for downloading an application on the board in S32 Design Studio differs according to the used MCU board. For questions, refer to the S32 Design Studio user guide. To download and debug on the S32K144 MCU board, follow the steps below:

1. Select the arrow next to the debug icon in the toolbar and select **Debug Configurations**, see Figure 23.



aaa-065429

Figure 23. Downloading the application (a)

2. In the **Debug Configurations** dialog box, select **S32K144_SDK_Project_Debug** under **GDB PEMicro Interface Debugging**.
3. Ensure that **C/C++ Application** contains path **to.elf** file of the project, see Figure 24.

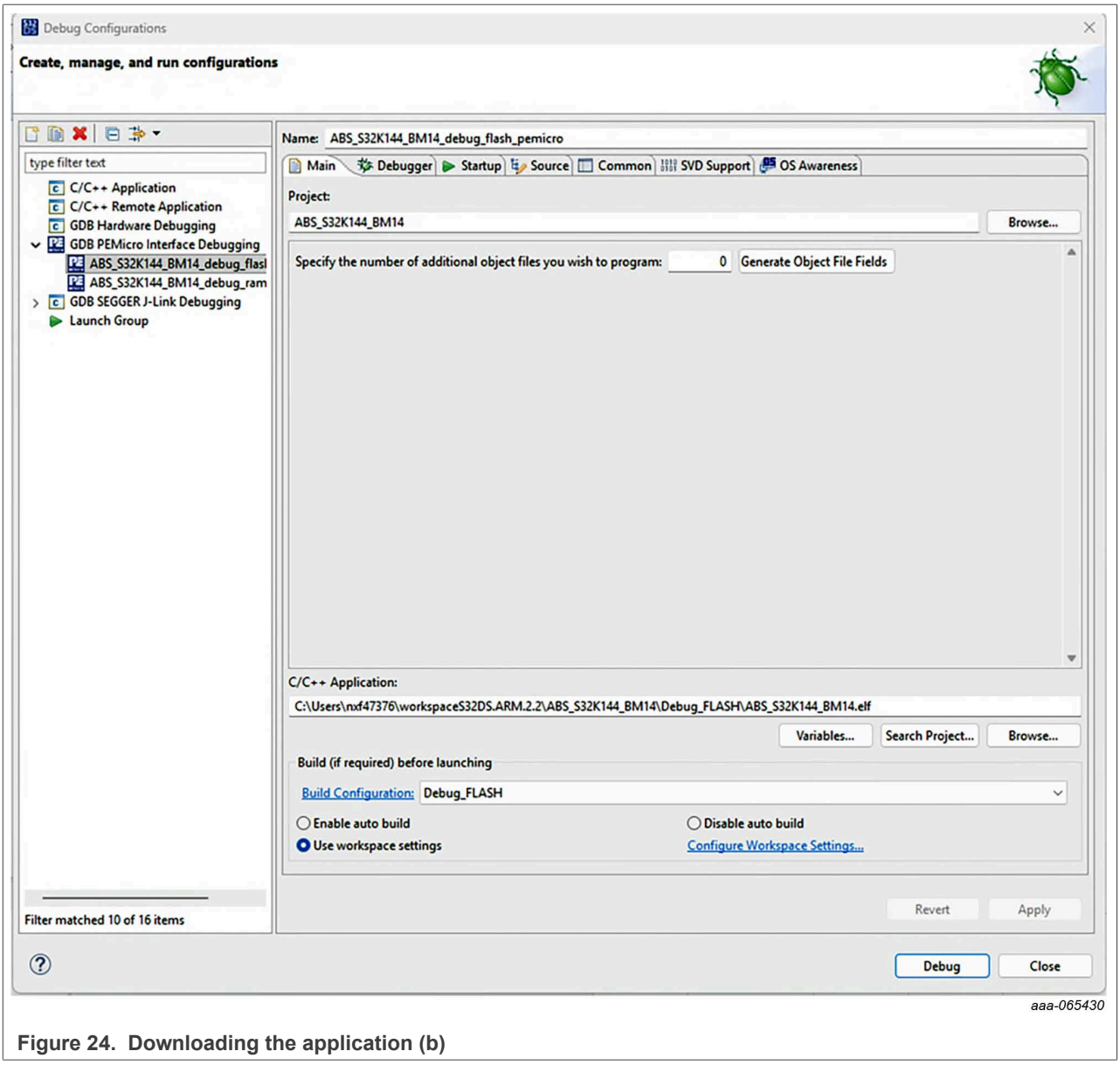
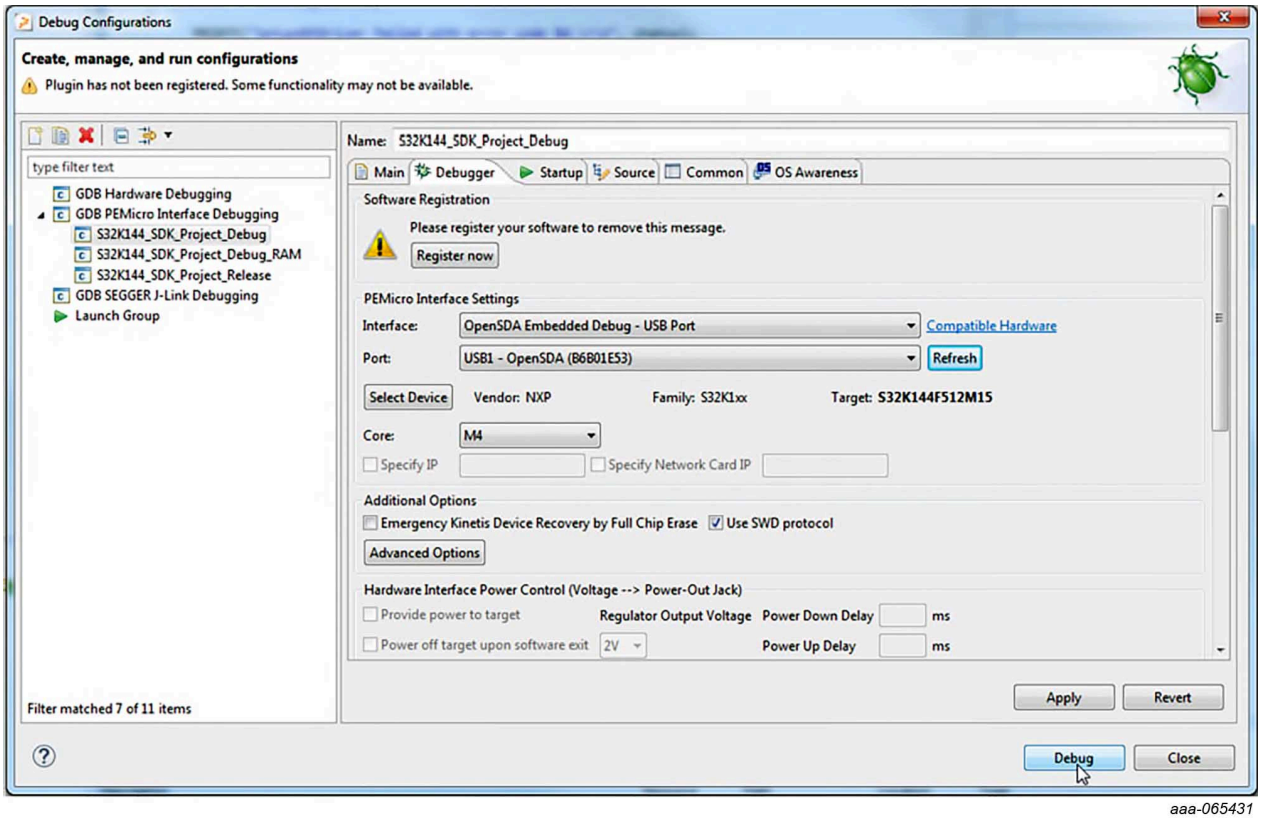


Figure 24. Downloading the application (b)

4. Select the **Debugger** tab and set the **Interface** option to **OpenSDA Embedded Debug – USB Port**. Then select the **Refresh** button next to the Port setting to update list of available USB ports, see Figure 25.
5. Ensure that the **Target** is set to **S32K144F512M15**. If not, change the target with use of the **Select Device** button. Click the button, in **Select Target Device** dialog box go to **NXP > S32K1xx > S32K144F512M15** and confirm with **Select** button.
6. Then select **Debug**. S32 Design Studio will download and launch the program on board.



aaa-065431

Figure 25. Downloading the application (c)

5 Abbreviations

Table 6. Software environment

Acronym	Description
AML	analog middleware layer
API	application programming interface, software interface for controlling functions
GUI	graphical user interface, visual user interface for software interaction
KDS	kinetis Design Studio Integrated development environment
SDK	software development kit, software package with drivers and examples
S32 DS	S32 Design Studio Integrated development environment
S32 SDK	S32 Software development kit (also called Auto SDK)

Table 7. Analog part

Acronym	Description
CSB	chip select bar to indicate active-low
DOSV	digital output supply voltage, configurable voltage level for digital I/O (e.g. 3.3 V / 5 V)
GPIO	general purpose I/O input/output
HD	high-side driver for fail-safe switch for overall solenoid path
HS	high side, driver controlling loads connected to battery side
LDD	low-side driver, driver that sinks current to ground for loads (e.g. solenoids)
LFSR	linear feedback shift register, logic for pseudo-random sequence generation or diagnostics
LPSPI	low power serial peripheral interface, low-power SPI module in MCU
LSD	low-side driver for inductive load
PD	pump motor pre-driver
PWM (PDI)	pulse width modulation (pump driver input), signal to control pump motor speed
RSTB	reset bar, active-low reset pin (LOW resets device)
SBC	system basis chip, IC combining power supply and communication functions for ECU
SCLK	serial clock, provides clock signal for SPI communication
SI	serial input, data input from MCU to SB0400 (SPI command input)
SO	serial output, data output from SB0400 to MCU (SPI response)
SPI	serial peripheral interface, communication protocol between MCU and SB0400
TPM / FTM	timer/PWM module / flexTimer module, MCU peripherals for timing and PWM generation
WLAMP/WLD	warning lamp
WLD	warning lamp driver, output driver for ABS warning lamp
WSAI	wheel speed analog input, analog input for wheel speed sensors
WSOx	wheel speed output, digital outputs related to wheel speed sensing
WSS	wheel speed sensor interface

6 Revision history

Table 8. Revision history

Document ID	Release date	Description
UG 10399 v.2.1	10 June 2026	• Added Section 3.4
UG 10399 v.2.0	01 March 2026	• Update with S32DS_ARM2.2 version
UG 10399 v.1.0	10 September 2016	• Initial version

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Suitability for use in automotive applications — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Figures

Contents