ARM Board Support Package Criticality B Qualification

Board Support Package – Software User Manual

ARMB-N7S-BSP-SUM rev. 1.4

N7 SPACE SP. Z O.O.

Prepared by	Date and Signature
Konrad Grochowski	Konrad Grochowski 2024.08.13 18:06:22 +02'00'
Verified by	
Mateusz Dyrdół	Mateusz Dyrdół 2024.08.13 18:32:04 +02'00'
Approved by	
Michał Mosdorf	Michał Mosdorf Digitally signed by Michał Mosdorf Date: 2024.08.13 20:37:50 +02'00'

ESA Contract No. 4000137041/22/NL/AS/kk

Table of Contents

1 Introduction 5
2 Applicable and reference documents 6
2.1 Applicable documents 6
2.2 Reference documents 6
3 Terms, definitions and abbreviated terms 7
4 Conventions 8
5 Purpose of the Software 9
6 External view of the software 12
7 Operations environment 13
7.1 General 13
7.2 Hardware configuration 13
7.3 Software configuration 13
7.4 Operational constraints 13
8 Operations basics 15
9 Operations manual 16
10 Reference manual 17
10.1 Introduction 17
10.2 Help method 17
10.3 Screen definitions and operations 17
10.4 Commands and operations 17
10.5 Error messages 17
11 Tutorial 18
11.1 Introduction 18
11.2 Getting started 18
11.2.1 Obtaining the source 18
11.2.2 Using the build environment in Docker 18
11.2.3 Building the drivers 19
11.2.4 Integrating BSP with user project 20
11.2.5 Recommended compilation options 21
11.3 Using the software on a typical task 21
11.3.1 Example program – LED blink 21
11.3.2 Asynchronous operations 25
11.3.3 RTEMS integration 29
11.3.4 BSP in MPLAB environment 32

ESA Contract No. 4000137041/22/NL/AS/kk

12 Analytical Index 38
13 Lists 39
13.1 List of Annexes 39
13.1.1 Annex A – Error codes 39
13.2 List of Tables 39
13.3 List of Figures 39
13.4 List of Listings 39

ESA Contract No. 4000137041/22/NL/AS/kk

Change Record

Issue	Date	Change
1.0	2023-07-10	Initial release
1.1	2023-10-12	Updates for QAR: • All BSP static libraires will be prefixed with n7s-bsp- • Some N/A justifications added • RTEMS integration manual added • Referenced documents updated
1.2	2023-11-16	Updates for QAR RIDs: • Fixed missing cross-link to Listing 24 • Contract number added to the footer • Referenced documents updated
1.3	2024-03-13	Updated for v4.3.1: • Added more information to chapter 11.3.3 (RTEMS integration layer), including examples description • Referenced documents updated
1.4	2024-08-13	Updated for v5.0.0: • Extending the deliverables to support SAMRH707F18 • Added missing n7s-bsp- to some examples • Described the presence of the VERSION file in the source code package • Described complete build sequence for all docker images required to build docker image from scratch

ESA Contract No. 4000137041/22/NL/AS/kk

1 Introduction

This document provides Software User Manual for the Board Support Package (BSP) deliverables of the ARM Board Support Package Criticality B Qualification project.

Board Support Package contains the low-level drivers for the peripherals of the SAMV71, SAMRH71 and SAMRH707 microcontrollers and validation test suites for qualification of those drivers.

The Software User Manual is produced as a standalone document and structured according to the SUM Document Requirements Definition (DRD) given in Annex H of ECSS-E-ST-40C [AD1].

ESA Contract No. 4000137041/22/NL/AS/kk

2 Applicable and reference documents

2.1 Applicable documents

ID	Title	Reference	Rev.
AD1	ECSS – Space engineering Software	ECSS-E-ST-40C	6 March 2009

2.2 Reference documents

ID	Title	Reference	Rev.
RD1	ARM Board Support Package Criticality B Qualification Board Support Package – Interface Control Document	ARMB-N7S-BSP-ICD	1.7
RD2	ARM Board Support Package Criticality B Qualification Board Support Package – Software Design Document	ARMB-N7S-BSP-SDD	1.8
RD3	ARM Board Support Package Criticality B Qualification Board Support Package – Software Configuration File	ARMB-N7S-BSP-SCF	1.8
RD4	ARM Board Support Package Criticality B Qualification Board Support Package – Coding Standards and Tools	ARMB-N7S-BSP-CSTD	1.5
RD5	Atmel SAM V71 Xplained Ultra USER GUIDE	Atmel-42408C	Rev. C – 09/2015
RD6	Microchip Technology Inc. SAMRH71F20-EK Evaluation Kit User's Guide	DS50002910A	Rev. A – 09/2019
RD7	Microchip Technology Inc. SAMRH71-TFBGA-EB User Guide	DS50003449A	Rev. A – 11/2022
RD8	Microchip Technology Inc. SAMRH707F18-EK Evaluation Kit User's Guide	DS60001744B	Rev. B – 02/2022

ESA Contract No. 4000137041/22/NL/AS/kk

3 Terms, definitions and abbreviated terms

This document acronyms and abbreviations are listed here under.

API	Application Programming Interface
BSP	Board Support Package
HW	Hardware
ISR	Interrupt Service Routine / Interrupt Handler
N7S	N7 Space
RTEMS	Real-Time Executive for Multiprocessor Systems
SW	Software

ESA Contract No. 4000137041/22/NL/AS/kk

4 Conventions

This Software User Manual describes a software project, therefore it refers to various commands that can be executed in the terminal and it presents various source code fragments. In order to make those special blocks more readable, numerous style conventions are used. This chapter quickly summarizes said conventions.

Short commands and code fragments that are embedded inside normal text paragraphs use `this style with a monospace font`.

Commands that are a bit longer or span multiple lines follow the following style:

```
$ command
Output (optional)
```

All commands listed in this manual were prepared and validated on Ubuntu 22.04 system. Although any similar Linux system should support all of the commands used in this document, it is recommended to use Ubuntu/Debian family.

Directory contents listings follow the same convention:

```
environment/
└── subfolder/
    └── file
lib/
└── a generic comment about contents of lib/
resources/
```

Source code blocks use the below style:

```
if (!Pio_init(LED_PIO_PORT, pio, errCode))
    return false;
```

The syntax highlighting colours used in the above block are defined as follows:

```
C Preprocessor directive
C Preprocessor include path
C Preprocessor definition
C Preprocessor symbol
Built-in types
User defined types
Function definitions
Function calls
Variable declaration
Struct members
Keywords
Number literals
Comments
Other
```

5 Purpose of the Software

The Board Support Package (BSP) is a set of low-level drivers for peripherals of the SAMV71, SAMRH71 and SAMRH707 microcontrollers. Each driver is provided as a C library, to be used by user's software to access and control the specific device. Table 1 lists all devices supported by the BSP.

The developed software is independent from any other library or operating system and fulfils ECSS Criticality Category B requirements.

Table 1 – BSP drivers list.

Drivers		**SAMV71**	**SAMRH71**	**SAMRH707**
Name	**Description**			
Adc	Analog-to-Digital Controller (ADC) device driver.			✓
Afec	Analog Front-End Controller (AFEC) device driver.	✓		
Dacc	Digital Analog Converter Controller (DACC) device driver.	✓		✓
Eefc	Enhanced Embedded Flash Controller (EEFC) device driver.	✓		
Flexcom	Flexible Serial Communication Controller (FLEXCOM) device driver.		✓	✓
FlexramEcc	FlexRAM Memory and Embedded Hardened ECC Controller (FLEXRAMECC) device driver.		✓	✓
Fpu	Floating Point Unit (FPU) device driver.	✓	✓	✓
Gmac	Ethernet (GMAC) device driver.	✓	✓	
Hefc	Hardened Embedded Flash Controller (HEFC) device driver.		✓	✓
Hemc	Hardened External Memory Controller (HEMC) device driver.		✓	✓
Hsdramc	Hardened SDRAM Controller (HSDRAMC) device driver.		✓	
Hsmc	Hardened Static Memory Controller (HSMC) device driver.		✓	✓
Isi	Image Sensor Interface (ISI) device driver.	✓		
Lpow	Low-power modes (LPOW) device driver.	✓		
Matrix	Bus Matrix (MATRIX) device driver.		✓	✓
Mcan	Controller Area Network (MCAN) device driver.	✓	✓	✓
Mpu	Memory Protection Unit (MPU) device driver.	✓	✓	✓
Nvic	Nested Vectored Interrupt Controller (NVIC) device driver.	✓	✓	✓
Pio	Parallel Input/Output Controller (PIO) device driver.	✓	✓	✓
Pmc	Power Management Controller (PMC) device driver.	✓	✓	✓
Pwm	Pulse Width Modulation Controller (PWM) device driver.	✓	✓	✓
Qspi	Quad Serial Peripheral Interface (QSPI) device driver.	✓		
Rstc	Reset Controller (RSTC) device driver.	✓	✓	✓
Rswdt	Reinforced Safety Watchdog Timer (RSWDT) device driver.	✓		
Rtc	Real-time Clock (RTC) device driver.	✓	✓	✓
Rtt	Real-time Timer (RTT) device driver.	✓	✓	
Scb	System Control Block (SCB) device driver.	✓	✓	✓
Sdramc	SDRAM Controller (SDRAMC) device driver.	✓		
Spi	Serial Peripheral Interface (SPI) device driver.	✓	✓	✓
Spw	SpaceWire (SPW) device driver.		✓	✓
Supc	Supply Controller (SUPC) device driver.	✓	✓	✓
Systick	System timer (SYSTICK) device driver.	✓	✓	✓
Tcm	Tightly Coupled Memory (TCM) device driver.		✓	✓
Tic	Timer Counter (TC) device driver.	✓	✓	✓

ESA Contract No. 4000137041/22/NL/AS/kk

Drivers		**SAMV71**	**SAMRH71**	**SAMRH707**
Name	**Description**			
Twihs	Two-wire Interface (TWIHS) device driver.	✓	✓	
Uart	Universal Asynchronous Receiver Transmitter (UART) device driver.	✓	✓	✓
Wdt	Watchdog Timer (WDT) device driver.	✓	✓	✓
Xdmac	DMA Controller (XDMAC) device driver.	✓	✓	✓

The drivers provide an interface to perform operations specific for each peripheral on the user side, while on the hardware side the communication focuses on the configuration of specific registers. Figure 1 presents an example of the BSP deployment as a component of the user's software.

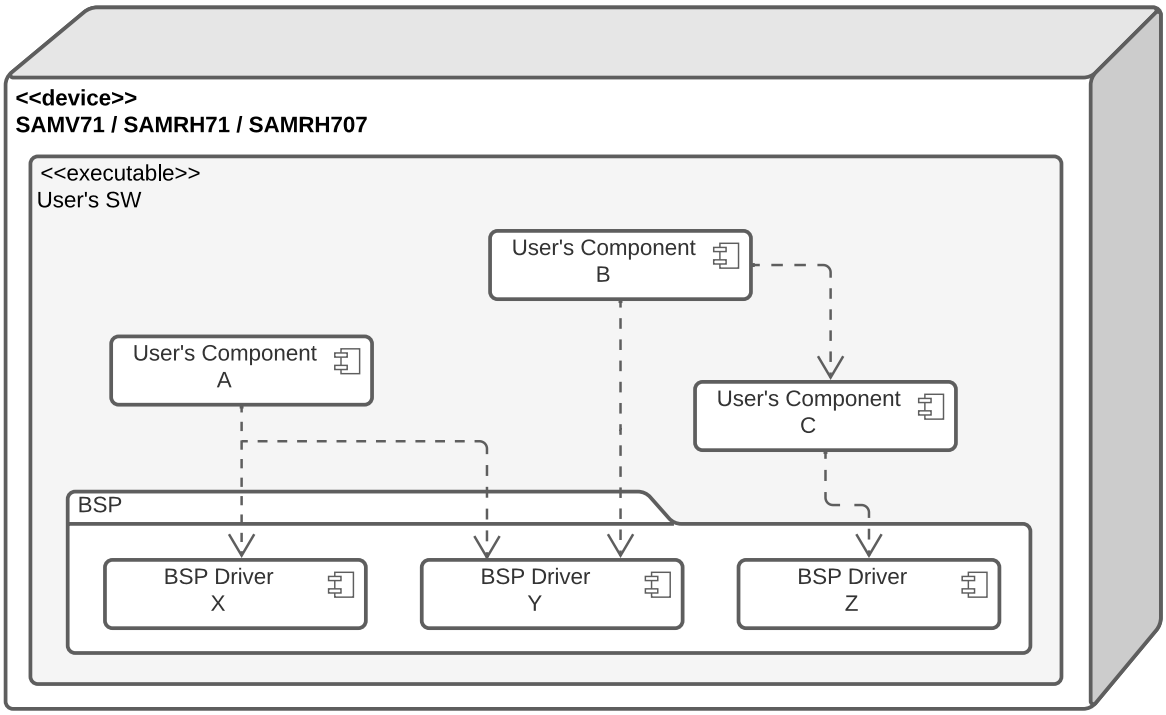

Figure 1 – BSP deployment example.

Each driver is a façade for the MCU registers and interrupts as seen on generic driver representation on Figure 2. Provided API should be concise, less error prone and more convenient then a direct manipulation of a various bits.

ESA Contract No. 4000137041/22/NL/AS/kk

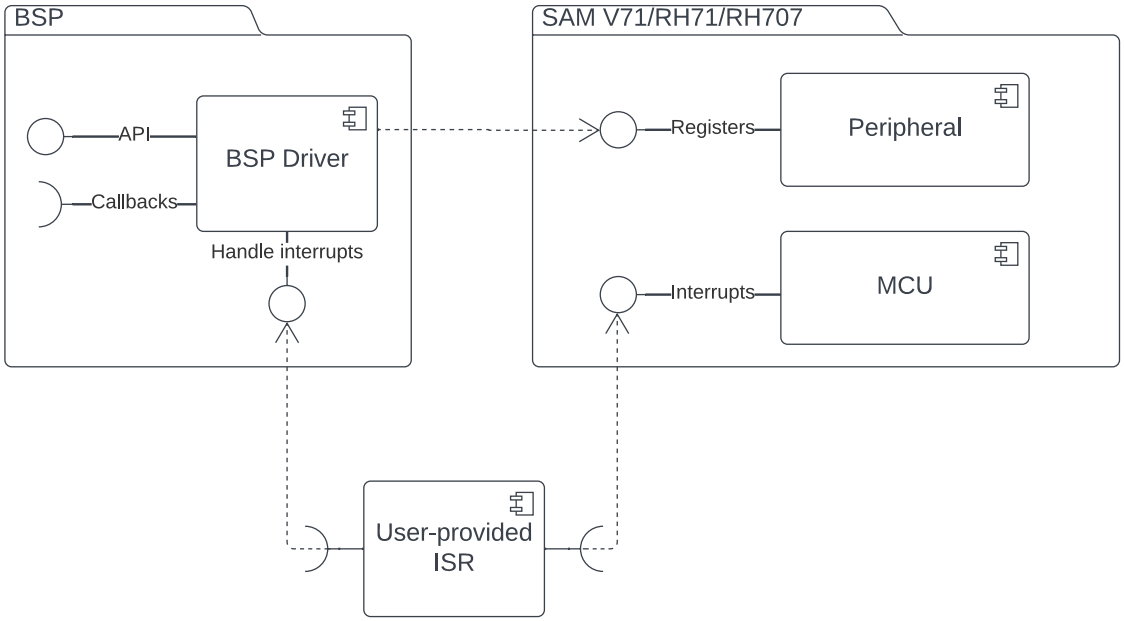

Figure 2 – Generic driver design.

Notice, that the driver does not interact directly with the MCU interrupts – this allows user to integrate the BSP with any operating system specific ISRs or use BSP in bare-metal implementation.

As an example of such integration, the BSP provides API layer for the drivers to be used as part of the RTEMS operating system. RTEMS API will be implemented using “adapter” design pattern, as shown on Figure 3. The RTEMS operating system provides some requirements on the drivers that could be used as elements of “RTEMS BSP” and the adapters will provide this interface, using BSP Driver as a implementation backend.

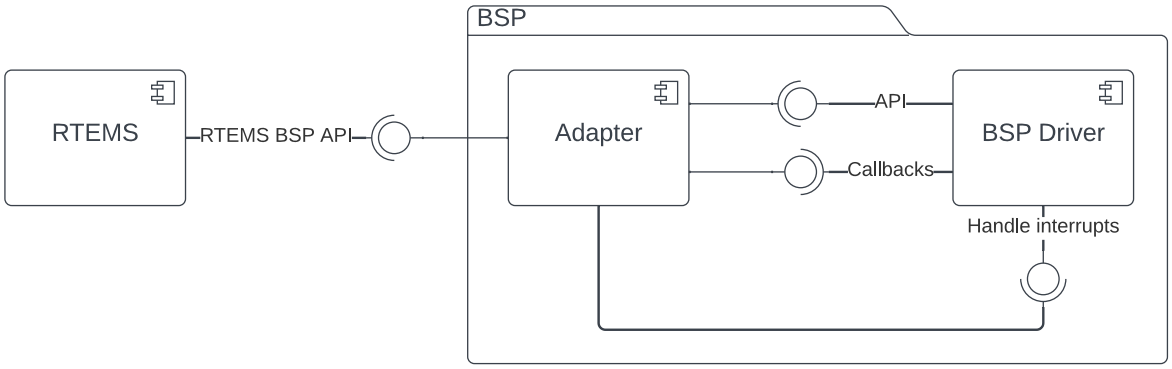

Figure 3 – Generic RTEMS Adapter design.

ESA Contract No. 4000137041/22/NL/AS/kk

6 External view of the software

BSP is delivered as an archive consisting of source files and SCons-based build system. The software itself consists of separate library for each of the provided drivers.

The directory structure can be described as follows (for clarity reduced to the most important items):

```
bsp/
├── doc/
│   └── Doxygen configuration file
├── environment/ - build and test environment
│   ├── configs/
│   │   └── SVF configuration files
│   ├── docker/
│   │   └── Docker image configuration
│   ├── ld/
│   │   └── linker scripts used by unit and integration tests
│   ├── lib/
│   │   └── test support libraries (startup, runtime, etc.)
│   ├── SpwRelay/
│   │   └── tool for handling SpaceWire Brick
│   └── TestFramework/
│       └── BSP specific parts of integration tests Python framework
├── lib/ - main source code directory (should be used as include path root)
│   └── n7s/
│       └── bsp/
│           └── XYZ/ - XYZ driver folder (generic layout, applied to each driver)
│               ├── tests/
│               │   └── unit tests of the driver
│               ├── XYZ.h - driver interface (C header file)
│               └── SConscript
├── resources/
│   ├── configs/
│   │   └── default BSP configuration files ("chip config")
│   └── n7-core/
│       ├── lib/ - utility libraries used across BSP,
│       └── environment/
│           └── test support C libraries base, integration tests Python framework
├── site_scons/
│   └── build system and integration testing utilities
├── tests/ - integration tests
│   └── XYZ/ - XYZ test folder (generic layout, applied to each test)
│       ├── bin/
│       │   └── source code of the C program used in the test
│       ├── SConscript
│       └── test_XYZ.py - test definition file
├── validation/
│   └── validation configuration (specifications, results, etc.)
├── README.md
└── SConstruct
```

The Software Configuration File [RD3] contains a detailed list of files in the library package along with their SHA-256 checksums.

ESA Contract No. 4000137041/22/NL/AS/kk

7 Operations environment

7.1 General

The BSP is designed to be included and used by other software. Each driver is a separate. Only a C compiler is required to build the libraries, and a C++ compiler to build unit tests. Libraries depend only on a basic subset of the C standard library. Example implementation of this subset is provided as a part of the test environment.

Build environment is described in [RD4] and documented in Docker image configuration in source code.

Note: although each driver is a separate library and does not depend on other libraries as a piece of software, there might be hardware dependency, that requires some peripherals to be configured together. The BSP assumes that it is user responsibility to configure subset of peripherals – this way user is free to use set of BSP drivers or mix BSP drivers with other setup code etc.

7.2 Hardware configuration

The BSW is a set of drivers for selected platforms: SAMV71Q21, SAMRH71F20 and SAMRH707F18. No special additional requirements are imposed on the user software. Memory usage or processor performance depends on the mission specific deployment of the BSP and should be checked by the end-user.

In the project the following development boards were used:

- Microchip ARM SAMV71Q21 microcontroller embedded in Microchip SMART SAM V71 Xplained Ultra (ATSAMV71-XULT) evaluation kit board described in [RD5].
- Microchip ARM SAMRH71F20 rad-hard microcontroller embedded in Microchip SAMRH71F20-EK evaluation kit board described in [RD6].
- Microchip ARM SAMRH707F18 rad-hard microcontroller embedded in Microchip SAMRH707F18-EK evaluation kit board described in [RD8].

Additionally, partial support for Microchip SAMRH71F20-TFBGA-EK evaluation board [RD7] is provided as tests tailoring options.

7.3 Software configuration

Refer to Figure 1 for example of BSP deployment. Each used library should be incorporated into final image of the user software (static linking).

If asynchronous (interrupt based) features of the BSP are used, user needs to provide a layer for integrating operating-system specific ISR with calls to BSP drivers.

7.4 Operational constraints

BSP is separated from operating system concerns and does not perform any internal synchronization to avoid data races. User should ensure that no BSP drivers methods are called from multiple threads/tasks on the same shared data or user should provide adequate synchronization techniques.

In case user wants to reconfigure working driver, special care needs to be taken regarding disabling interrupts before changing data shared with user-provided ISR.

ESA Contract No. 4000137041/22/NL/AS/kk

When using BSP on SAMRH707 platform, it's important to know about the undocumented issue with DMA access via AHBS port of the core, which causes only 32-bit accesses to be supported. This can cause memory-related issues when peripherals that use DMA (either explicitly, or implicitly – like MCAN, SpW or GMAC) have their memory buffers stored in memory only accessible via AHBS port, like DTCM. Therefore, it's recommended to make sure that DMA-accessible buffers are placed in other memories, like SRAM.

ESA Contract No. 4000137041/22/NL/AS/kk

8 Operations basics

N/A – The software in this project is designed to be included and used by other software. Therefore there are no predefined operational tasks. Staffing concerns, standard daily operations and contingency operations are all dependent on the final software based on BSP.

ESA Contract No. 4000137041/22/NL/AS/kk

9 Operations manual

Operations manual is not provided for BSP as justified in previous chapter.

ESA Contract No. 4000137041/22/NL/AS/kk

10 Reference manual

10.1 Introduction

A complete reference manual of the programming interfaces of each of the modules of BSP is available as the Doxygen-generated documentation supplied with SDD [RD2] Annex A. It is generated from source code of the Library and inline comments written for every public API function. Doxygen-style comments in all public header files used for generation of the reference manual can also be inspected directly.

General description of the driver interface layout is provided in the ICD [RD1].

Coding convention, standard and tools are described in [RD4].

Commands listed in the following chapters assume Linux host – preferably Ubuntu 22.04 or similar.

10.2 Help method

Each BSP public function is documented with a basic description, the meaning of each input parameter and return value, and a reminder on how to access error information in case of failure. This information is available in the `Doxygen`-generated documentation and in the header files themselves.

The unit-tests of each driver can be treated as function-per-function documentation by example.

Integration tests of drivers can serve as examples of complete programs using the drivers.

While building the BSP, the `scons` tool have built-in help describing available options:

```
$ scons -H # provides help for the SCons tool itself
$ scons -h # provides help for the BSP compile options
```

10.3 Screen definitions and operations

N/A – no graphical user interface or operations in the project.

10.4 Commands and operations

N/A – no commanding in the software (driver library).

10.5 Error messages

Each BSP function that can report an error returns `bool` (`true` on success, `false` on failure) and accepts optional pointer to variable of `ErrorCode` type as the last argument. In case of the failure (and if the pointer is not `NULL`) the specific error code will be written at the provided memory. Error codes are driver specific and list of all possible error codes is provided in 13.1.1 – Annex A – Error codes.

The BSP uses assertions to validate input arguments. It is recommended to enable assertions for development and disable them for release – they detect possible integration errors.

ESA Contract No. 4000137041/22/NL/AS/kk

11 Tutorial

11.1 Introduction

This tutorial serves as an introduction to the Board Support Package for SAMV71Q21, SAMRH71F20 and SAMRH707F18 platforms. Its goal is to demonstrate how to use the provided API to perform basic tasks related to the peripherals operations. For simplicity the examples covers only some of the drivers, but presented general approach applies to all drivers. Means to obtain detailed reference for each driver is described in chapter 10.

This tutorial assumes a basic level of knowledge of the hardware platforms and only provides an introduction to the BSP, written specifically for software engineers – potential users of the drivers.

11.2 Getting started

11.2.1 Obtaining the source

BSP source can be obtained by extracting delivered ZIP archive as in Listing 1.

Listing 1 – Unpacking BSP source from ZIP file.

```
$ unzip ARMB-N7S-BSP-SRC-v5_0_0.zip  # assuming version 5.0.0
```

Or (recommended option on Linux as BSP uses symbolic-links) from TAR BZIP2 - Listing 2.

Listing 2 – Unpacking BSP source from TAR BZIP2 file (recommended for Linux).

```
$ tar -xvf ARMB-N7S-BSP-SRC-v5_0_0.tar.bz2  # assuming version 5.0.0
```

Source archive contains `VERSION` file, which provides information about release of the package. It is used by the build system for producing reports. Build system can also use direct `git` commands to obtain version information, if the source code is put under configuration control using that tool. This allows for better tracking of reports for specific modification of the source. To use this feature, the `VERSION` file must be removed from the source code.

11.2.2 Using the build environment in Docker

Using Docker is the easiest way to reproduce necessary software environment. Otherwise user needs to install the dependencies from [RD4], using operating-system specific packages, which is out of the scope of this document. The minimal set consists of SCons build tool and ARM GCC compiler, but executing tests or performing static analysis of the code requires more dependencies to be installed.

Docker environment is distributed with build environment in a form of Docker image.

Listing 3 uses the Docker image provided as deliverable (it might take minutes to perform the import).

Listing 3 – Importing BSP build environment Docker image.

```
$ docker image load --input ARMB-N7S-BSP-ENV-v5_0_0.tar.bz2 # assuming version 5.0.0
Loaded image: n7s/arm/bsp:v5.0.0
```

Alternatively, image can be built „from scratch" (assuming all packages are still available) using Dockerfile provided in BSP source, as in Listing 4. Note that two images are built in that listing – one is a preliminary image ("core") which is not used later, but required to build the primary image itself.

Listing 4 – Building BSP Docker image.

```
# assuming version 5.0.0 in the commands below
$ cd <path/to/bsp/source>/resources/n7-core/environment/docker
$ docker build -t n7s/n7-core/n7-core:arm-bsp-v5.0.0 .
$ cd <path/to/bsp/source>/environment/docker
$ docker build --build-arg REGISTRY=n7s \
               --build-arg=BASE_IMAGE_TAG=arm-bsp-v5.0.0 \
                -t n7s/arm/bsp:v5.0.0 .
```

After setting up the image, user might use Docker containers as in Listing 5.

Listing 5 – Executing command in BSP build environment Docker container.

```
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) n7s/arm/bsp:v5.0.0 <COMMAND>
```

This command will mount current directory and execute container with privileges of current user. It is recommended to call it this way always in the root of the BSP source directory.

It can be very convenient to set up this command as an alias in Linux shell as in Listing 6. This will allow for a quick execution of other commands inside containers.

Listing 6 – Shell alias for executing command in build environment Docker container.

```
$ alias docker-here='docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g)'
```

For example, to check correctness of the image and BSP source, user might execute commands like in Listing 7 (or without alias as in Listing 8) and expect similar output.

All following commands in this chapter assume that there are either executed on properly configured environment, or are proceeded with `docker run` alias.

Listing 7 – Example command executed in BSP build environment Docker container.

```
$ cd <path/to/bsp/source>
$ docker-here n7s/arm/bsp:v5.0.0 scons -h
scons: Reading SConscript files ...
# ...
# ... other help lines ...
# ...
Board Support Package for SAMV71Q21 and SAMRH71F20 - v5.0.0
Copyright N7 Space sp. z o.o. 2018-2024
# ...
```

Listing 8 – Example command executed in BSP Docker container.

```
$ cd <path/to/bsp/source>
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) n7s/arm/bsp:v5.0.0 scons -h
# same output as in Listing 7
```

11.2.3 Building the drivers

This section assumes that the BSP main directory is the current working directory. In order to build a static version of a selected driver in "debug" configuration, command from Listing 9 should be executed.

Listing 9 – Build tool configuration for building driver in *debug* mode (without optimalization).

```
$ scons build=debug checkCode=0 n7s-bsp-pio-samv71q21
```

In the listing the PIO driver for SAMV71Q21 platform was selected. The `checkCode=0` flag speeds up the build process by disabling static analysis tools. To build an optimized "release" variant (with the optimization flags set to `-O2`, other options include `Os` and `O1`), command from Listing 10 should be executed. Assertions can be kept for “release” build, but on the example they are disabled.

Listing 10 – Build tool configuration for building driver in *release* mode (with optimization).

```
$ scons build=release checkCode=0 optimization=2 disableAsserts=1 n7s-bsp-pio-samv71q21
```

Each driver can be built using `<peripheral>-<platform>` naming scheme. User can specify multiple drivers or even `bsp-<platform>` to build all available drivers for selected platform. Calling `scons` without specifying target is not supported. Omitting `-<platform>` part of the target will build it for both platforms.

After completion of the build command, the requested driver can be found in `build/<build type>/<platform>/install_root/lib/` folder, as shown on Figure 4.

```
build/
└── release/ - build type
    └── samv71q21/  - target platform
        └── install_root
            └── lib
                └── libn7s-bsp-pio.a - the driver library
```

Figure 4 – Build directory layout.

11.2.4 Integrating BSP with user project

To use the BSP in user project, following options need to be set in the target build system:

- Target platform compiler switch (set for the pre-processor/compiler).
- Include path (directory to be searched for the BSP header files, set for the compiler).
- Library path (directory to be searched for the built libraries, set for the linker).

Assuming `BSP` points to the root BSP directory, those are:

- Platform switch: `N7S_BSP_<PLATFORM>` (e.g. `N7S_TARGET_SAMV71Q21`)
- Include path: `<BSP>/lib`
- Library path: `<BSP>/build/<build type>/<platform>/install_root/lib`

Listing 11 shows an example of compiling user file using GCC with include path and SAMV71Q21 platform configured.

Listing 11 – Compiling with GCC and include path example (BSP installed in `/opt/bsp`).

```
$ arm-none-eabi-gcc -c \
                    -I/opt/bsp/lib \
                    -DN7S_BSP_SAMV71Q21
                    -o user.o user.c # other compiler options
```

Listing 12 shows linking the user project using GCC linker (linking with PIO driver as an example).

ESA Contract No. 4000137041/22/NL/AS/kk

Listing 12 – Linking with GCC example (BSP compiled for SAMV71Q21 in `/opt/bsp`).

```
$ arm-none-eabi-gcc -L/opt/bsp/build/release/samv71q21/install_root/lib \
                    -ln7s-bsp-pio
                    -o user.elf \
                     user.o # other compiler options
```

11.2.5 Recommended compilation options

While compiling the user file it needs to be compiled in compatible way to the BSP. User might also want to compile the BSP outside of the SCons build system (to integrate the BSP into larger project).

Listing 13 shows the recommended compilation options (without optimization options) to compile BSP and user files for the target platform.

Listing 13 – Compilation options for SAMV71Q21/SAMRH71F20/SAMRH707F18 using GCC.

```
$ arm-none-eabi-gcc --std=c99 \
                    -mlittle-endian \
                    -mthumb \
                    -mcpu=cortex-m7 \
                    -mfloat-abi=hard \
                    -mfpu=fpv5-d16 \
                    -Dasm=__asm__ \
                     # other options
```

The include path and pre-processor related options listed in 11.2.4 also needs to be included.

11.3 Using the software on a typical task

11.3.1 Example program – LED blink

The example program presented in this chapter will introduce the user to basic concepts of working with the BSP drivers. The example describes a program that blinks one of the diodes available on the development boards.

Program starts with the inclusion of BSP headers for drivers of required modules (Listing 14):

- PIO – Parallel Input/Output Controller – to control the state of the pin connected to the diode,
- PMC – Power Management Controller – to enable power on the pin port,
- WDT – Watchdog Timer – to disable the watchdog timer.

Listing 14 – LED example – includes.

```
#include <n7s/bsp/Pio/Pio.h>
#include <n7s/bsp/Pmc/Pmc.h>
#include <n7s/bsp/Wdt/Wdt.h>
```

Note the complete path of the includes – see 11.2.4 for details on include path setting. This path reduces the chance of conflict on header files' names.

The development boards for SAMV71Q21, SAMRH71 and SAMRH707F18 have diodes connected to different pins. Listing 15 presents platform-specific definitions to be used in the example.

ESA Contract No. 4000137041/22/NL/AS/kk

Listing 15 – LED example – pre-processor defines.

```
#if defined(N7S_TARGET_SAMV71Q21)
#define LED_PIO_PORT   Pio_Port_A
#define LED_PIN_MASK   PIO_PIN_23
#define LED_PMC_PERIPH Pmc_PeripheralId_PioA
#elif defined(N7S_TARGET_SAMRH71F20)
#define LED_PIO_PORT   Pio_Port_B
#define LED_PIN_MASK   PIO_PIN_19
#define LED_PMC_PERIPH Pmc_PeripheralId_PioA
#elif defined(N7S_TARGET_SAMRH707F18)
#define LED_PIO_PORT   Pio_Port_B
#define LED_PIN_MASK   PIO_PIN_11
#define LED_PMC_PERIPH Pmc_PeripheralId_PioA
#else
#error "Missing N7S_TARGET_* macro"
#endif
```

Note the `N7S_TARGET_` macro being used – this macro is necessary for proper integration of the BSP headers, so it as well can be used by the example itself.

As the processors start with watchdog enabled, a procedure for disabling it might be useful for simplest examples (it is not recommended to disable watchdog in critical software). Listing 16 presents a basic watchdog disable procedure. It is a common BSP use pattern – initialize a driver using its `_init` procedure and then configure the peripheral using `_setConfig` call. Note that although only `isDisabled` field is needed to be set in the configuration structure, it is filled with all values – this is a recommended approach to ensure consistent driver and peripheral setup

Listing 16 – LED example – disable watchdog.

```
static void disableWatchdog(void)
{
  Wdt wdt;
  Wdt_init(&wdt);
  const Wdt_Config config = {
    .isDisabled = true,
    .isFaultInterruptEnabled = false,
    .isResetEnabled = false,
    .counterValue = 0xFFFu,
    .deltaValue = 0xFFFu,
    .isHaltedOnDebug = false,
    .isHaltedOnIdle = false,
#ifndef N7S_TARGET_SAMV71Q21
    .doesFaultActivateProcessorReset = true,
#endif
  };
  Wdt_setConfig(&wdt, &config);
}
```


ESA Contract No. 4000137041/22/NL/AS/kk

Drivers provide just an interface to the peripheral registers, so they do not need to be de-initialized or explicitly destroyed. This also means that further in the code another `Wdt` object could be initialized and used to reconfigure the same WDT module. It is recommended though to use the same object through all the calls in the program, as some drivers might have additional internal state. In the example the WDT will never again be manipulated by the program, hence the `Wdt` object can be a stack variable.

The next procedure in the example, shown on Listing 17, is responsible for configuration of the PIO and PMC modules so they will allow for diode manipulation.

Listing 17 – LED example – PIO configuration.

```
static bool initPio(Pio* const pio, ErrorCode* const errCode)
{
  if (!Pio_init(LED_PIO_PORT, pio, errCode))
    return false;

  Pmc pmc;
  Pmc_init(&pmc, Pmc_getDeviceRegisterStartAddress());
  Pmc_enablePeripheralClk(&pmc, LED_PMC_PERIPH);

  const Pio_Pin_Config pinConfig = {
    .control = Pio_Control_Pio,
    .direction = Pio_Direction_Output,
    .pull = Pio_Pull_None,
#ifdef N7S_TARGET_SAMV71Q21
    .filter = Pio_Filter_None,
    .isMultiDriveEnabled = false,
    .irq = Pio_Irq_None,
    .driveStrength = Pio_Drive_Low,
#else
    .isOpenDrainEnabled = false,
    .irq = Pio_Irq_EdgeFalling,
    .isIrqEnabled = false,
    .driveStrength = Pio_Current_2m,
#endif
    .isSchmittTriggerDisabled = false,
  };

  return Pio_setPinsConfig(pio, LED_PIN_MASK, &pinConfig, errCode);
}
```

Function signature follows the pattern from the BSP – it returns Boolean indication of success and has additional argument for detailed error code. The error code pointer is passed down to BSP function calls. Notice, that although `Pio` driver configures the PIO controller, the explicit configuration of the PMC module is required. The relation between PIO port and PMC peripheral is fixed and could be automatically handled by the BSP, but the drivers are by design independent – `Pio` driver interacts with PIO and PIO only. This design trade-off allows for mixing the drivers with other means of platform configuration (different drivers, direct register manipulation etc.). It also solves some issues when the order of various peripherals configuration needs to be controlled by the user. It is up to the user to determine the set of modules that need to be configured to achieve required functionality.

Next, the example contains an abstraction layer for LED control – shown on Listing 18 (the development boards have different meaning of HIGH state for LED control).

Listing 18 – LED example – LED on/off procedures.

```
static void setLedOn(Pio* const pio)
{
#ifndef N7S_TARGET_SAMV71Q21
  Pio_setPins(pio, LED_PIN_MASK);
#else
  Pio_resetPins(pio, LED_PIN_MASK);
#endif
}

static void setLedOff(Pio* const pio)
{
#ifndef N7S_TARGET_SAMV71Q21
  Pio_resetPins(pio, LED_PIN_MASK);
#else
  Pio_setPins(pio, LED_PIN_MASK);
#endif
}
```

Mentioned abstraction layer is than used to implement crude blinking mechanism shown on Listing 19.

Listing 19 – LED example – blink procedure.

```
static void crudeDelay(const uint32_t delay)
{
  for (uint32_t i = 0; i < delay; i++)
    asm volatile("nop" ::: "memory");
}

static void blinkLed(Pio* const pio, const uint32_t delay)
{
  setLedOn(pio);
  crudeDelay(delay);
  setLedOff(pio);
  crudeDelay(4u * delay);
}
```

Finally, the `main` procedure is presented on Listing 20. Notice the comment showing where the possible error handling procedure could be located. The code returned from the function could be logged or used to determine specific recovery action. The pattern used through the BSP allows to easily gather or types of errors from various drivers on various levels into single handling procedure, or to create multiple procedures that even pass unhandled errors to higher levels etc.

ESA Contract No. 4000137041/22/NL/AS/kk

Listing 20 – LED example – *main* procedure.

```
int main()
{
  disableWatchdog();

  ErrorCode errCode = ErrorCode_NoError;
  Pio pio;
  if (!initPio(&pio, &errCode))
    return (int)errCode; // or any other processing

  setLedOff(&pio);

  while (true)
    blinkLed(&pio, 500000u);
}
```

The full code of this example is available inside the source code archive in `examples/LedBlinker` folder. Binary image of the example can be built using `led-blinker.elf` target passed to SCons.

For further working examples please refer to the code of unit and validation tests. Those present the proper order of the modules configuration and common use case examples.

11.3.2 Asynchronous operations

Various peripherals of the SAMV71Q21/SAMRH71F20/SAMRH707F18 support performing operations executed asynchronously to the program instruction processing. For example executing transmission using external interfaces etc. The BSP provides two ways of handling such asynchronous operations: by *polling* the status of the peripheral or by *callbacks* called from interrupts processing routines. The availability of each of the methods depends on the hardware capability and can vary between drivers, but whenever possible BSP delivers both.

Figure 5 presents generalized sequence of the polling mechanism. The user code calls the required methods of the driver to start the asynchronous operation and then continues to check (poll) the status of the peripheral.

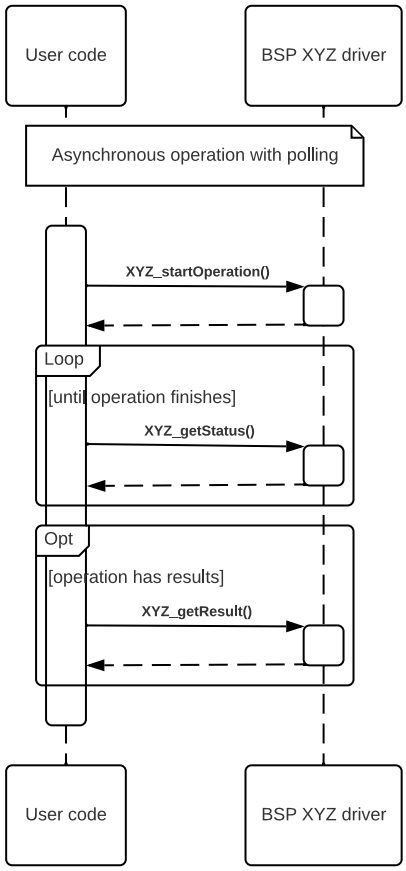

Figure 5 – Asynchronous operation with polling.

Listing 21 presents basic example using UART driver. The method in the example calls `Uart_write` procedure to send a single byte over UART interface and then polls on `Uart_isTxEmpty` until the transmission is finished. Notice, that BSP provides its internal polling mechanism, that allows to sequence multiple `Uart_write` calls – each will wait for the peripheral to become available before writing. In the example code waits for `Uart_isTxEmpty` in the loop, but user of course can implement it differently, checking the state from some operating system scheduled task or even not checking it at all until the status of the peripheral is required elsewhere.

Listing 21 – Asynchronous operations example – polling.

```
static bool uartPolling(Uart* const uart, ErrorCode* const errCode)
{
  const uint8_t data = 0x42u;

  if (!Uart_write(uart, data, UART_TIMEOUT, errCode))
    return false;

  while (!Uart_isTxEmpty(uart))
  {
    // polling
  }
  // operation finished
}
```


ESA Contract No. 4000137041/22/NL/AS/kk

Figure 6 presents the generalized callback-based approach for handling asynchronous task. The BSP provides a method that can be passed a *handler* – a pair of pointer to function (*callback*) and user data to be passed to the pointed function. Those methods register the handler in internal BSP structures and it will be called from the interrupt handling procedure when the operation finishes. It is important to remember, that the handler function will be called from the context of the ISR – user needs to ensure proper synchronization of access to shared data (if needed). Also note, that the ISR itself is user-provided – this allows BSP to be used in different environment: bare metal or various operating systems, as long as the user provides proper implementation of the interrupt handling.

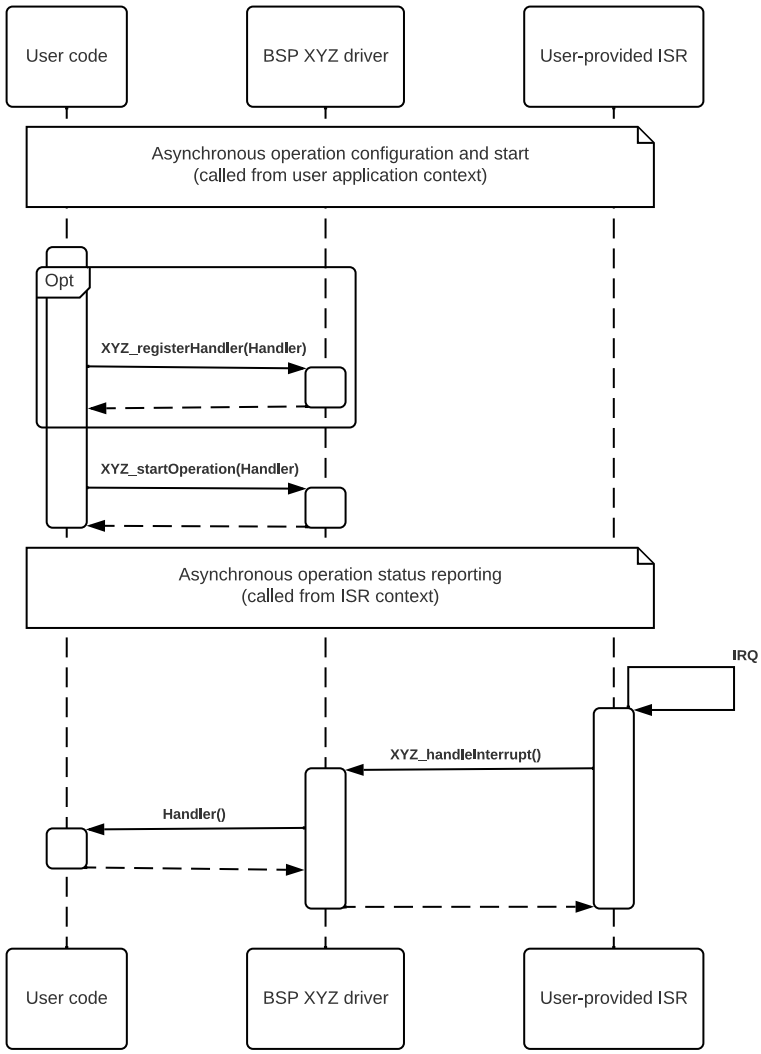

Figure 6 – Asynchronous operation with ISR callbacks.

Listing 22 presents an example of interrupt based transmission using UART. The user prepares `ByteFifo` (queue) with data to be sent and calls `Uart_writeAsync`, passing to it handler that is relying on the user specific data. The data passed to handler can be `NULL`, as it is not interpreted in any way by the BSP and only passed to user provided function. The function pointer itself also can be `NULL` – in such case user will not be informed about end of the transmission, but interrupt handling must still be implemented for BSP to properly process data queue. User can mix polling techniques to obtain status of the peripheral, but this is not a recommended approach as it might lead to data races – implementing the handler is a better approach when the moment of the end of the operation must be known.

ESA Contract No. 4000137041/22/NL/AS/kk

Listing 22 – Asynchronous operations example – callback.

```
struct UserData {
  uint8_t someData;
};

static ByteFifo* endOfTransmissionCallback(void* const data)
{
  struct UserData* user = (struct UserData*)data;
  user->someData = 42u;
  return NULL; // or next portion of data to transmit
}

static void uartHandler(void)
{
  const Uart_TxHandler handler = {
    .callback = endOfTransmissionCallback,
    .arg = &userData,
  };

  Uart_writeAsync(&uart, createByteFifo(), handler);
}
```

In some cases, like in the UART example, BSP handlers provide additional features, apart from simple informing about completion of the operation. As shown in the example, the handler can provide next queue to be sent, which transmission will be immediately started, without a need for second `Uart_writeAsync` call. Such approach allows for streamlining of the UART transmission.

Error handling in interrupt bases asynchronous operations requires registering additional handler. Some drivers accepts that driver in the method that start the operation itself, but in case of the UART a call to `Uart_registerErrorHandler` is required – see Listing 23. This is also a handler that will be called from the context of ISR processing.

Listing 23 – Asynchronous operations example – error handling.

```
static void errorCallback(const Uart_ErrorFlags* errorFlags, void* arg)
{
  struct UserData* user = (struct UserData*)arg;
  if (errorFlags->hasParityErrorOccurred)
    user->someData = 0u; // user choice of error handling/notification
}

static void registerErrorHandler()
{
  const Uart_ErrorHandler handler = {
    .callback = errorCallback,
    .arg = &userData,
  };

  Uart_registerErrorHandler(&uart, handler);
}
```

The last necessary part of the interrupt based processing that user needs to provide is the ISR implementation itself. Listing 24 shows a simple *bare metal* implementation (as used in unit and validation tests of the BSP).

Example in Listing 24 implements a ISR by following a common ARM naming conventions which allows the linker to assign the ISR to proper location in interrupt vector. `Uart_handleInterrupt` is called by the handler and inside it BSP will perform all necessary UART related operations and if needed call user provided handlers. Notice that in case of the bare metal implementation, as seen on the listing, it is the user's responsibility to clear pending interrupts (in this case `Nvic` driver from the BSP was used). The example is provided in Listing 24.

Listing 24 – Asynchronous operations example – ISR implementation.

```
void UART4_Handler(void)
{
  ErrorCode errCode = ErrorCode_NoError;
  if (!Uart_handleInterrupt(&uart, &errCode))
  {
    // process error code
  }
  Nvic_clearInterruptPending(Nvic_Irq_Uart4);
}
```

Important difference that can be noticed between the polling and interrupt based examples is about lifetime and visibility of the `Uart` driver object in the code. In case of the polling based example, the object can be passed to the function and its lifetime only needs to last to the end of the function. In case of the interrupt based code, the object lifetime must last at least to the end of the ISR processing. The object itself must also be accessible by the ISR – in case of the presented bare metal implementation it needs to be a static variable (global to the file). Special care must be taken by the user when dealing with this kind of global object and lifetime of the objects during asynchronous operations.

For further working examples please refer to the code of unit and validation tests. Those present the proper order of the modules configuration and common use case examples.

11.3.3 RTEMS integration

BSP can be used with RTEMS operating system and applications built with RTEMS. Adapters and configuration files necessary to build RTEMS with BSP are provided. BSP start hooks use functions from Pmc and Scb BSP modules to configure processor clocks and cache according to selected tailoring. Minimal set of BSP modules needed to compile RTEMS includes:

- Systick
- Rtc
- Pmc
- Uart (+ Flexcom on RH71 and RH707)
- Pio
- Nvic
- Scb
- Utils
- Eefc (for V71)

RTEMS integration layer provides drivers and adapters for several BSP components that can be used through RTEMS API functions. Systick module is used as RTEMS scheduler clock and should not be manipulated by user directly. Rtc module can be used through RTEMS clock manager API, Uart selected

in tailoring config is used as RTEMS simple console, rest of Uarts available on microcontroller can be used directly (through Uart/Flexcom BSP modules) for other purposes.

If other BSP modules or other source files that support RTEMS BSP are required they can be added to `bsp_modules` array in script `build-rtems.sh` and their sources to `rtems_build_spec/n7bsp/obj.yml` if they should be compiled for all platforms or to `rtems_build_spec/n7bsp/target-*.yml` if they should be compiled only for a selected target. There are several software prerequisites that need to be fulfilled to use BSP with RTEMS:

1. Working RTEMS toolchain (*gcc* with *newlib* and *binutils*),
2. Compatible RTEMS source distribution (version from SMP QDP is compatible),
3. RTEMS source builder (RSB),
4. WAF build tool,
5. RTEMS WAF support library,
6. RTEMS Tools.

All tools mentioned above can be downloaded and installed directly by the user or obtained using either pre-built Docker image or Docker image built by the user using provided Dockerfile.

To build Docker image with tools needed to compile RTEMS and BSP:

```
$ cd <path/to/rtems-bsp/source>/environment/docker
$ docker build -t n7s/arm/rtems-bsp:v5.0.0 .  # assuming version 5.0.0
```

Resulting Docker image contains RTEMS source code, RSB, RTEMS WAF, and RTEMS tools repositories checked out to commits matching RTEMS SMP QDP. Image also contains RTEMS toolchain built using RSB. Copy of WAF build system tool is provided in RTEMS source code repository. Before building RTEMS system itself user can tailor RTEMS and BSP configuration. Basic RTEMS configuration is stored in files (one file per each platform):

```
rtems_build_spec/config-*.ini
```

Compiler flags and scope of build can be adjusted there.

BSP configuration is stored in two places, first one contains default configuration and should not be modified by the user (it can be used as a reference to determine which options are available and may require tailoring – defaults will work for development boards used in project):

```
resources/n7bsp/resources/configs/default-*.conf
```

Second set of BSP configuration files contains RTEMS or board-specific options and should be modified according to user preferences. Options from tailoring files can override default settings placed in matching section under matching names. Tailoring configuration is stored here:

```
src/bsps/n7sbsp/config/tailoring-*.conf
```

Important options that should be checked are clock and memory configuration.

Section `[rtems.bspinit]` contains `N7S_BSPINIT_CONFIGURE_PMC` switch (with default value 1) that changes behaviour of BSP initialization hook executed during application startup. Applications that are standalone – loaded into processor's internal flash memory or booted from external memory should leave this enabled and configure PMC module in order to run. Applications loaded by bootloader (for example BSW) to external memory (especially when executing code from SDRAM) should be compiled with `N7S_BSPINIT_CONFIGURE_PMC` option set to 0. Reconfiguration of PMC

settings that impact HEMC/HSDRAMC clock (for SAMRH71F20) or SDRAMC clock (for SAMV71Q21) when executing code from external SDRAM can lead to program crash and lockup due to processor being unable to fetch instructions. If application is required to change PMC settings it should execute configuration procedure from one of internal processor memories and make sure that SDRAM is accessible after clock changes. Note that settings specified in `[pmc]` section are applied only if `N7S_BSPINIT_CONFIGURE_PMC` is set to 1. Settings in `[core]` section must be set to correct values (left by the bootloader) even if PMC is not configured be application as they are used in Systick rate and UART baud rate calculations.

Second important tailoring option is memory configuration. Default tailoring file contains several pre-defined configurations that can be changed by adjusting `configuration` option in section `[rtems.memories]`. Default configuration is `intflash` that is useful for standalone applications booted from internal processor FLASH without the use of bootloader. Options `intsram` and `sdram` are for applications running from internal processor SRAM memory and external SDRAM respectively and they are compatible with use of bootloader that configures PMC and memory controller before launching application. Custom sets of memory region aliases can be added by the user by adding section `[rtems.memories.XYZ]` and setting `configuration = XYZ` in section `[rtems.memories]` where XYZ is the name of the configuration. New section should define aliases for all memory regions specified by RTEMS. Available memories are defined in file `rtems_build_spec/n7bsp/linkcmds.yml` and custom values can be added there if required.

Section `[uart]` contains settings for UART used as basic RTEMS console (standard input and output).

After configuration is adjusted, RTEMS system may be compiled by executing following command in main repository directory.

```
$ docker-here <image-with-rtems-toolchain> ./build-rtems.sh -p <platform>
```

where <platform> can be samv71q21, samrh71f20 or samrh707f18.

Bash script is provided for convenience and executes following procedure (in temporary docker container)

1. Copy RTEMS configuration file for selected platform into the container
2. Copy build spec files (YAML) for BSP into the container
3. Copy BSP adapters and drivers into the container
4. Copy selected BSP modules source code into the container
5. Invoke WAF build tool to configure and compile RTEMS system
6. Install compiled RTEMS system into directory outside Docker container (build-<platform>)
7. Modify pkg-config metadata file to point to installed RTEMS

After RTEMS is compiled, example applications can also be built:

```
$ docker-here <image-with-rtems-toolchain> ./build-apps.sh -p <platform>
```

where <platform> can be samv71q21, samrh71f20 or samrh707f18.

This command will invoke WAF build tool in temporary Docker container to compile example applications using previously built RTEMS and install them in directory build-apps-<platform>.

Source code of example applications is in directory *test_apps*.

There are 3 examples available

1. *n7_ticker* – Most basic example built for all platforms which creates two RTEMS tasks. One task sends event every 1/3rd of a second, other tasks receives the events and prints date and time obtained from RTC.
2. *n7_uart_example* – Also creates 2 tasks. One task sends event every 3 seconds. Other task configures UART and receives events from UART irq handler and first task. Every event from UART irq triggers received data processing, event from first task triggers transmission of current status using UART. Data processing is simple received byte counting and calculating sum of all received bytes excluding newline characters (`0x0A`).
3. *n7_spw_example* – Available only for SAMRH71F20 and SAMRH707F18 as it requires SpaceWire support. This example tests SpaceWire links connected in external loopback – it requires that loopback cable is connected between SpaceWire LINK1 and LINK2 available on microcontroller. There are 3 RTEMS tasks created: CTRL, TX and RX. RX and TX tasks have 2 data buffers each.
   - CTRL task sends start event to RX task and waits for initialization.
   - RX sets up buffer for reception and sends RX ready event to CTRL.
   - CTRL sends start event to TX task.
   - TX sets up buffers, send lists, starts transmission and sends TX started event to CTRL.
   - CTRL now listens for various events (packet transmitted/received, data verification failure, RX/TX complete).
   - RX listens for Buffer activated/deactivated events sent by SPW IRQ, upon reception of activated event next buffer is set up. Upon reception of deactivated event current buffer is verified (only 4 first bytes are checked, rest should be zero) in case of failure RX fail event is sent to CTRL. If maximum number of packets is received RX complete event is sent to CTRL. If terminate event is received task terminates.
   - TX listens for sendListActivated events sent by SPW IRQ. Upon reception of activated event next send list is set up. If maximum number of packets is transmitted TX complete event is sent to CTRL. If terminate event is received task terminates.
   - When CTRL receives TX and RX complete events it exits main loop and sends terminate event to RX and TX, prints test results and terminates.
   - SPW IRQ sends RX packet and TX packet event to CTRL on EOP reception and transmission respectively (used to count packets), it also sends sendList and buffer status updates to TX and RX to swap buffers when needed.

   By default 1024 data packets are transmitted (each consisting of 16384 bytes) which translates to 16 MiB of data for SAMRH71 and 16384 packets (each consisting of 4096 bytes) which translates to 64 MiB of data for SAMRH707. Number of packets and their sizes can be customized by editing system.h file in example directory. Program calculates and prints statistics after performing the test. When running from integrated FlexRam memory it should achieve data transmission speeds close to theoretical maximum for SpaceWire link (160 Mbps) for SAMRH71 and about 106 Mbps for SAMRH707. Transmission rate for SAMRH707 is slower due to usage of smaller packets – processing overhead is bigger. Smaller packets are required to fit in available memory.

11.3.4 BSP in MPLAB environment

The BSP can be integrated with Microchip MPLAB IDE in two ways:

1. by linking the pre-built objects of the BSP,
2. by using BSP MPLAB project.

The first option guarantees that the target project relies on the same object files as the one that passed the tests and other verification activities. But it might not be the most convenient one, especially when debugging is necessary or some modifications of the BSP are foreseen. Hence the other option – the BSP provides an MPLAB project which allows it to be opened inside MPLAB X IDE, with all needed source files available. Then such project can be connected to user target project and built together. But user caution is necessary – the BSP built using MPLAB compiler *was not tested.* The MPLAB compiler and compiler used by the BSP are compatible, but to keep the pre-qualification status it is recommended to use the first mentioned MPLAB integration approach when creating the final product.

It is worth reading 11.2.4 first, to get the general requirement on reusing the BSP inside user's project. The LedBlinker example (available in BSP source package and described in 11.3.1) can be used to check user's MPLAB project configuration.

11.3.4.1 Using pre-built BSP in MPLAB

Unpack the `ARMB-N7S-BSP-OBJ` and `ARMB-N7S-BSP-SRC` in a folder accessible by MPLAB. It's recommended to put those files on the same mount point/hard drive as target projects files, since MPLAB may have issues with using them if they are somewhere else.

In the MPLAB project that will use BSP:

1. Add unpacked libraries to the project.
    - Right-click the project in Projects tree and open its Properties (Figure 7).
    - Go to "Libraries" section and click "Add Library/Object File..." (Figure 8).
    - Select required libraries and add them to the project.

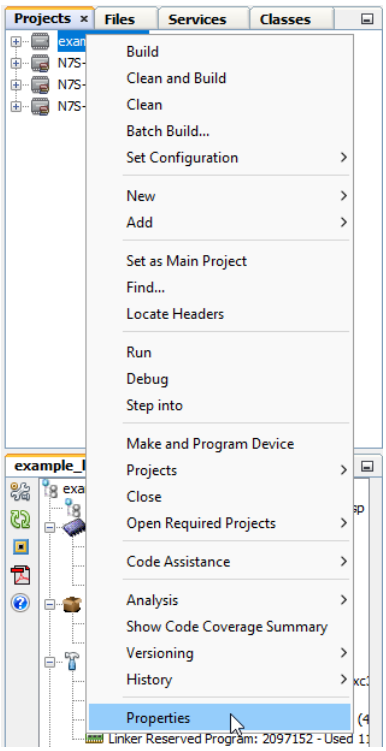

Figure 7 – Accessing MPLAB project's properties.

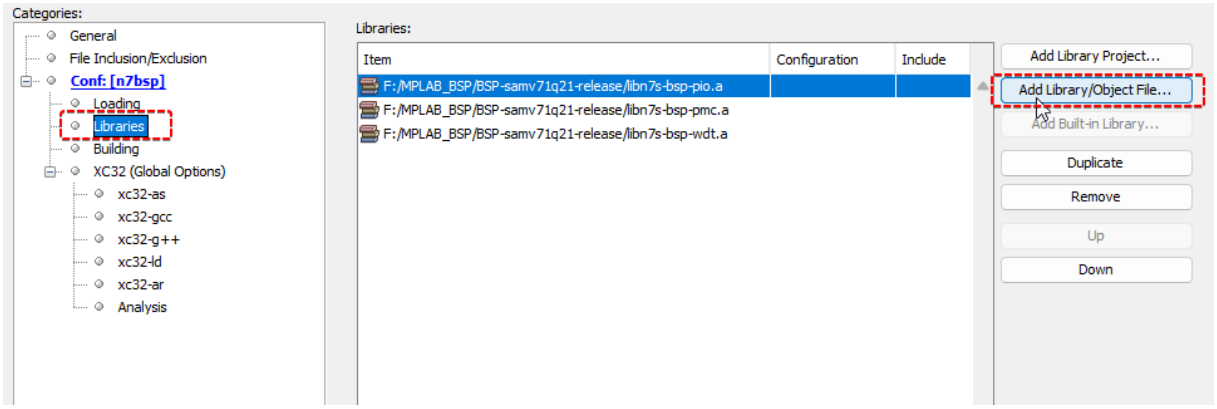

Figure 8 – Adding libraries to MPLAB project.

2. Add to the include directories: `lib` and `resources/n7-core/lib` subfolders of the SRC.
   - Go to "XC32 (Global Options)" and click on "Common include dirs" option (Figure 9).
   - Add paths with BSP headers to the list (Figure 10).

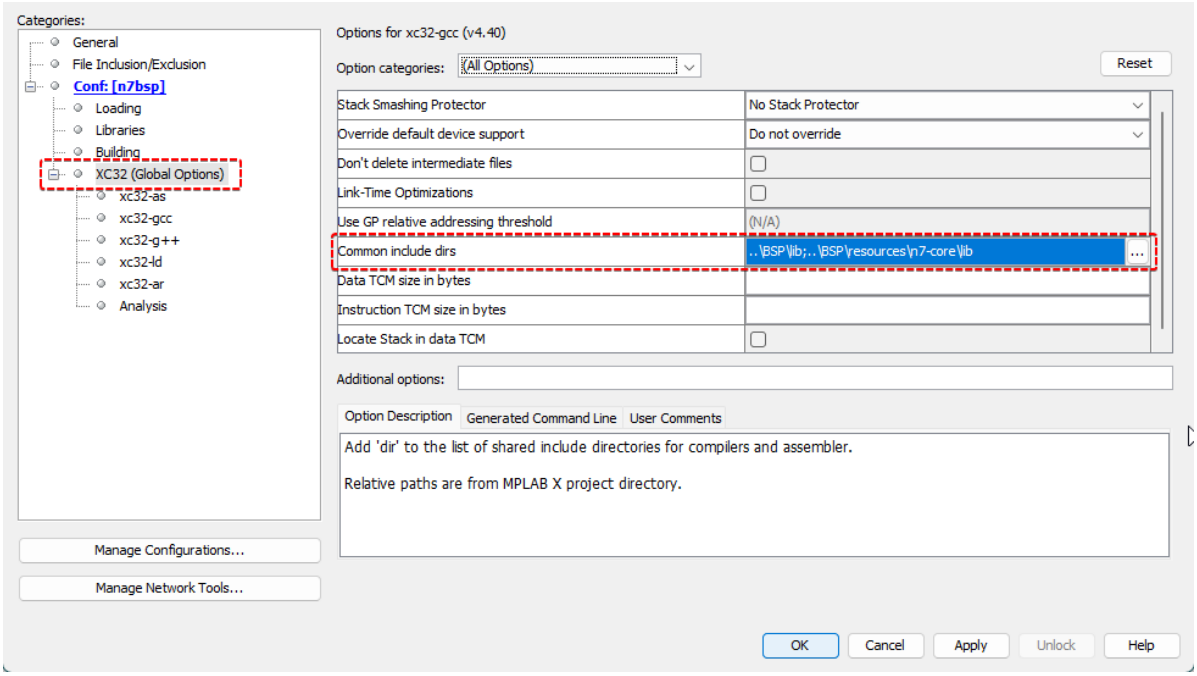

Figure 9 – MPLAB project properties.

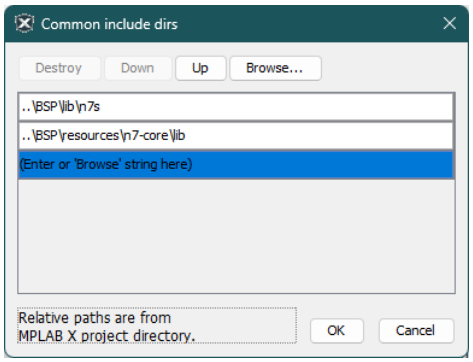

Figure 10 – MPLAB project's include directories list (BSP as standalone library).

ESA Contract No. 4000137041/22/NL/AS/kk

3. Add required `N7S_TARGET_<MCU>` macro to preprocessor directives (see also 11.2.4).
    - Go to target compiler's settings (xc32-gcc for C, xc32-g++ for C++) and for each compiler:
        - Select "Preprocessing and messages" in Option categories (Figure 11).
        - Click on "Preprocessor macros" field and add the macros to the list.

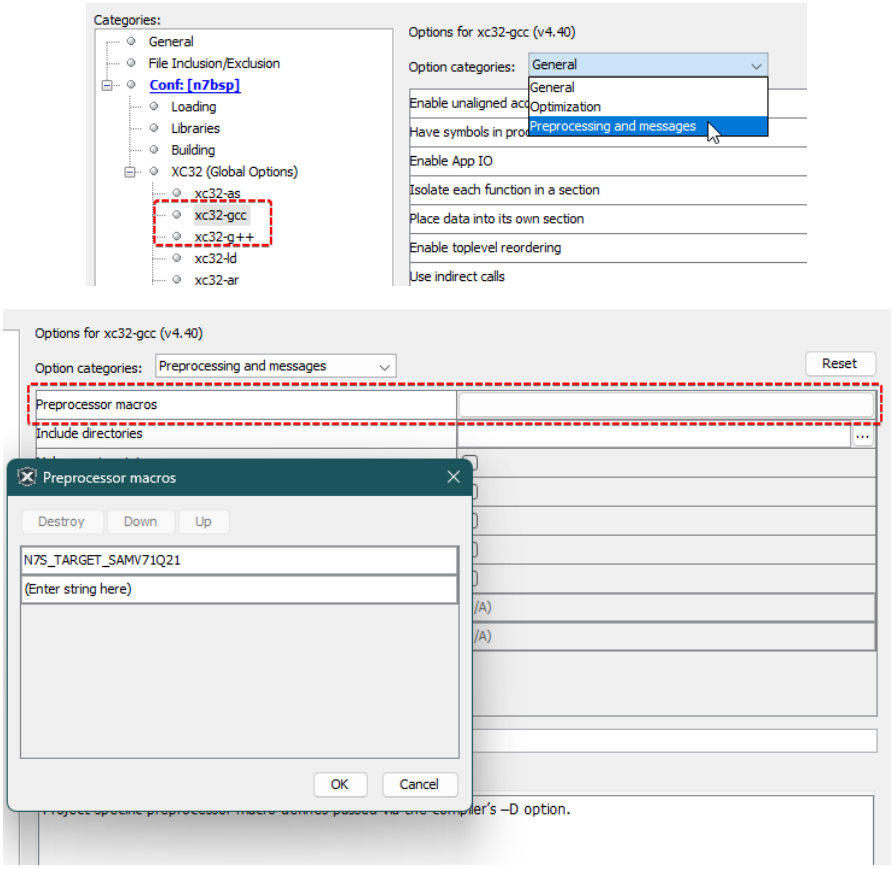

Figure 11 – MPLAB project's preprocessor options.

11.3.4.2 Using BSP as MPLAB project

Note: read about the verification limitations listed in parent chapter (11.3.4).

The `ARMB-N7S-BSP-SRC` deliverable provides archive, that contains dedicated MPLAB project – one for each supported platform. User can unpack selected project and open it in MPLAB – it should provide complete BSP in a form of library project (Figure 12).

ESA Contract No. 4000137041/22/NL/AS/kk

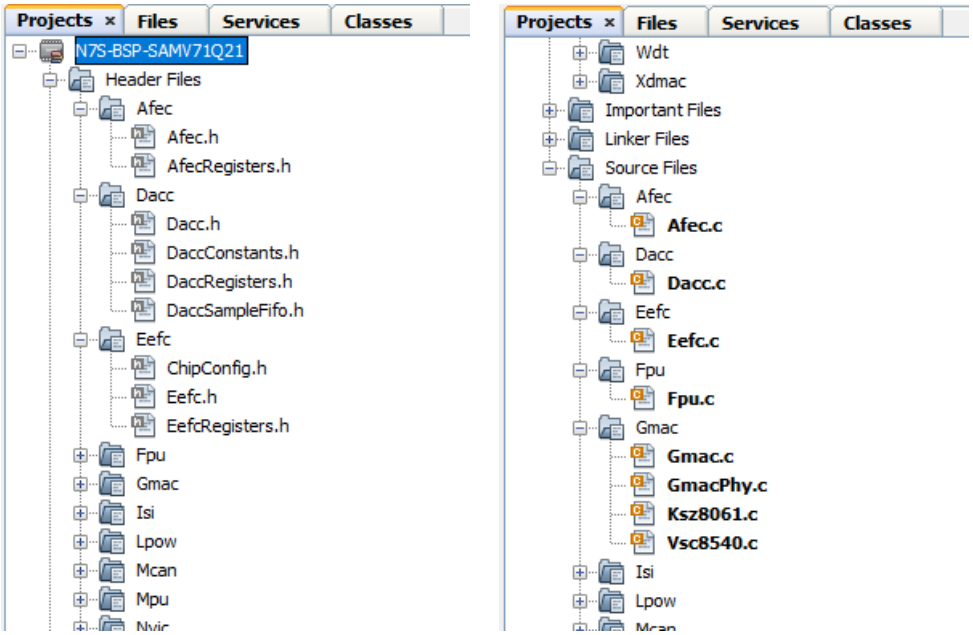

Figure 12 – BSP as MPLAB project.

Then, the user might choose to use BSP project as a dependency in a target project. This requires following steps:

1. Adding project as a dependency in a target project.
   - Right-click the project in Projects tree and open its Properties (Figure 7).
   - Go to “Libraries” section and click “Add LibraryProject...” (Figure 13).
   - Select BSP project and add them to the target project.

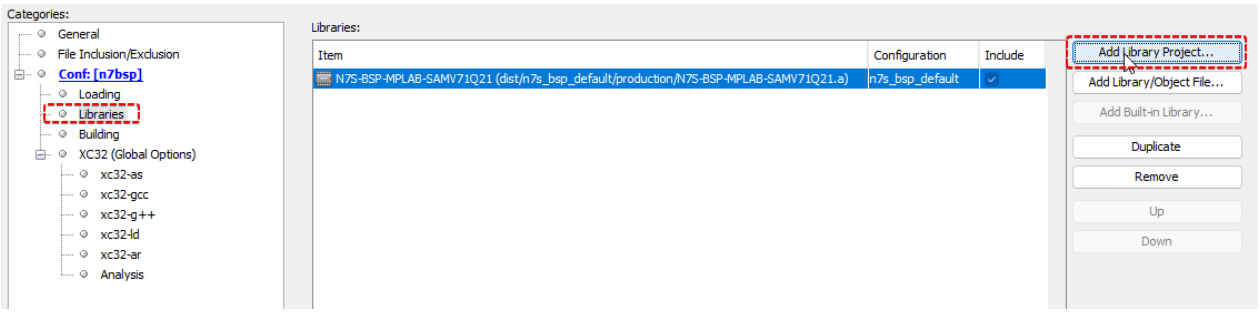

Figure 13 – Adding library project to MPLAB project.

2. Add to the include directories: `lib` subfolders from the BSP project.
   - Go to “XC32 (Global Options)” and click on “Common include dirs” option (Figure 9).
   - Add paths with BSP headers to the list (Figure 14).

ESA Contract No. 4000137041/22/NL/AS/kk

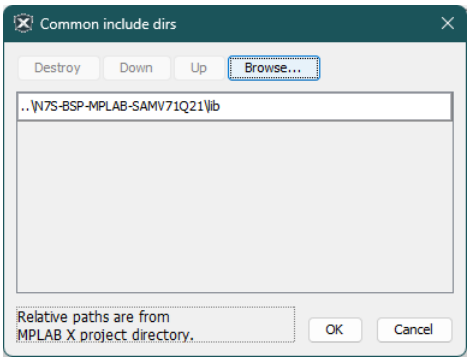

Figure 14 – MPLAB project's include directories (BSP as MPLAB project).

3. Add required `N7S_TARGET_<MCU>` macro to preprocessor directives (see also 11.2.4).
   - Go to target compiler's settings (xc32-gcc for C, xc32-g++ for C++) and for each compiler:
     - Select "Preprocessing and messages" in Option categories.
     - Click on "Preprocessor macros" field and add the macros to the list (Figure 11).

Then the target project can be built and BSP drivers used in it.

*Note:*if the target project uses Gmac module, the necessary Phy submodule must be selected. The BSP contains two implementations (used by various models of development boards of the supported platforms), one must be removed or the linker will complain about duplicated symbols.

In some cases user might want to generate MPLAB project manually from the BSP source code (e.g. the code or flags were modified using "standard" BSP development environment and now user would like to migrate to MPLAB). To do so, compile the BSP using steps from 11.2.3. This should produce `compile_commands.json` file for the selected platform inside `build` directory. Then from inside the BSP source folder, using environment prepared as described in 11.2 execute the command from Listing 25 (the listing assumes SAMRH71F20 platform). The last argument of the script is the path to the folder which will contain generated MPLAB project (if the folder exists, it will be cleared, all its contents will be removed). This script can also be used to generate project using debug configuration.

Listing 25 – Generating MPLAB project for the BSP (for RH71 – SAMRH71F20).

```
$ ./environment/mplab/generate_mplab_project.py \
                build/release/samrh71f20/compile_commands.json \
                RH71 \
                N7S-BSP-MPLAB-SAMRH71F20
```


ESA Contract No. 4000137041/22/NL/AS/kk

12 Analytical Index

N/A

ESA Contract No. 4000137041/22/NL/AS/kk

13 Lists

13.1 List of Annexes

13.1.1 Annex A – Error codes

Archive containing definitions of BSP internal error codes that can be reported by called functions using `ErrorCode` output parameter.

File name: `ARMB-N7S-BSP-SUM-A Error Codes.zip`
SHA256: `739c9721263ea4d0a44b353389685e360b4527562e646c4f143fc18a0265f455`

13.2 List of Tables

Table 1 – BSP drivers list. 9

13.3 List of Figures

Figure 1 – BSP deployment example. 10
Figure 2 – Generic driver design. 11
Figure 3 – Generic RTEMS Adapter design. 11
Figure 4 – Build directory layout. 20
Figure 5 – Asynchronous operation with polling. 26
Figure 6 – Asynchronous operation with ISR callbacks. 27
Figure 7 – Accessing MPLAB project's properties. 33
Figure 8 – Adding libraries to MPLAB project. 34
Figure 9 – MPLAB project properties. 34
Figure 10 – MPLAB project's include directories list (BSP as standalone library). 34
Figure 11 – MPLAB project's preprocessor options. 35
Figure 12 – BSP as MPLAB project. 36
Figure 13 – Adding library project to MPLAB project. 36
Figure 14 – MPLAB project's include directories (BSP as MPLAB project). 37

13.4 List of Listings

Listing 1 – Unpacking BSP source from ZIP file. 18
Listing 2 – Unpacking BSP source from TAR BZIP2 file (recommended for Linux). 18
Listing 3 – Importing BSP build environment Docker image. 18
Listing 4 – Building BSP Docker image. 19
Listing 5 – Executing command in BSP build environment Docker container. 19
Listing 6 – Shell alias for executing command in build environment Docker container. 19
Listing 7 – Example command executed in BSP build environment Docker container. 19
Listing 8 – Example command executed in BSP Docker container. 19
Listing 9 – Build tool configuration for building driver in *debug* mode (without optimalization). 20
Listing 10 – Build tool configuration for building driver in *release* mode (with optimization). 20
Listing 11 – Compiling with GCC and include path example (BSP installed in `/opt/bsp`). 20
Listing 12 – Linking with GCC example (BSP compiled for SAMV71Q21 in `/opt/bsp`). 21
Listing 13 – Compilation options for SAMV71Q21/SAMRH71F20/SAMRH707F18 using GCC. 21

Listing 14 – LED example – includes. 21
Listing 15 – LED example – pre-processor defines. 22
Listing 16 – LED example – disable watchdog. 22
Listing 17 – LED example – PIO configuration. 23
Listing 18 – LED example – LED on/off procedures. 24
Listing 19 – LED example – blink procedure. 24
Listing 20 – LED example – *main* procedure. 25
Listing 21 – Asynchronous operations example – polling. 26
Listing 22 – Asynchronous operations example – callback. 28
Listing 23 – Asynchronous operations example – error handling. 28
Listing 24 – Asynchronous operations example – ISR implementation. 29
Listing 25 – Generating MPLAB project for the BSP (for RH71 – SAMRH71F20). 37

ESA Contract No. 4000137041/22/NL/AS/kk