mikroSDK

PROGRAMMER'S MANUAL

VERSION 1.0

December, 2017.

MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

TABLE OF CONTENTS

PREFACE

This document explains the mikroSDK as the **software development kit** used for the portable application code development, but also as the implementation of the mikroSDK **standard**.

The mikroSDK standard describes all the components of the mikroSDK, along with a set of rules that must be followed to ensure that the future improvement and extensions of the mikroSDK will be compatible with the already developed content. The aim of the mikroSDK standard is to establish a well-defined form of content, which will be provided as a software support for the existing click boards™ product line. This standard also defines rules that must be followed, so that the further extension of the mikroSDK is possible, without breaking the backward compatibility.

Libraries that are part of the mikroSDK and that are developed according to the mikroSDK standard can easily integrate future extensions such as the additional architectures and development systems, without any or with minimal changes of the previously developed implementation.

This document also contains additional descriptions regarding the coding rules, key functions documentation, the application code, descriptions of the additional files included - and the package format.

The mikroSDK is composed of a number of various libraries and function calls, divided into two separate layers:

- click libraries
- mikroBUS API

The click library layer is strictly focused on the click board™ itself. The click board™ can be equipped with a wide range of different sensors, actuators, displays and other types of integrated circuits.

The mikroBUS API is focused on development systems and microcontrollers used for development and prototyping. It covers a wide range of different architectures and vendors.

Both layers must stay compatible and future improvements must be strictly defined, according to the mikroSDK standard.

NOTE

Each mikroSDK layer must be capable to carry extensions of any other layer. Each new extension will not break the backward compatibility.

In addition to these two layers, provided demo application code can be considered as the additional, third layer. The purpose of the demo application code is to demonstrate the basic functionality of a certain sensor, actuator or any other device for which the code is developed. The developers can use those simple demo applications as a starting point for their own application development. Also, some parts of the demo application such as the configuration variables, are defined by the mikroSDK standard.

SUMMARY

The main goal of the mikroSDK standard is to establish a set of rules so that the provided solution has a strictly defined form and structure. This allows for the code to be reusable and independent of used hardware platform or MCU architecture, as long as it stays mikroSDK standard compliant.

ANALYSIS

The structure of the software provided with the various click boards™ so far, can be divided into two main parts: the library and the example code *(Figure 1)*.

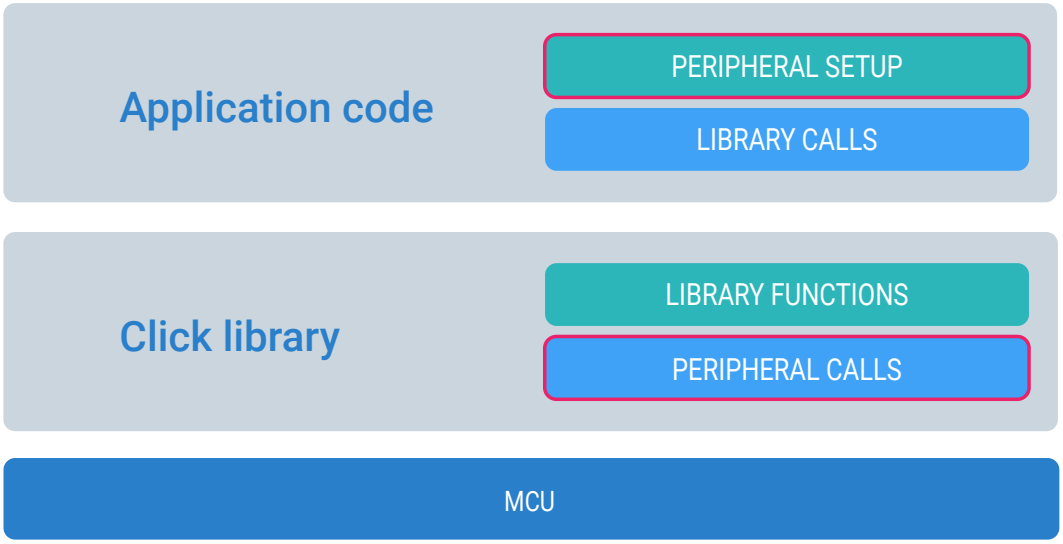


Figure 1: The structure of the usual click board™ supporting software

The diagram in Figure 1 shows two main parts and their subparts. The marked subparts represent an architecture-dependent code, which means that the code for each of these marked subparts should be redeveloped for every new architecture and development system.

The mikroSDK standard solves the problem of code redevelopment by making each of

the architecture-dependent parts of the library and the application code - reusable. The purpose of the standard is to prescribe a set of rules, which place all the architecture dependent code inside the specific layers.

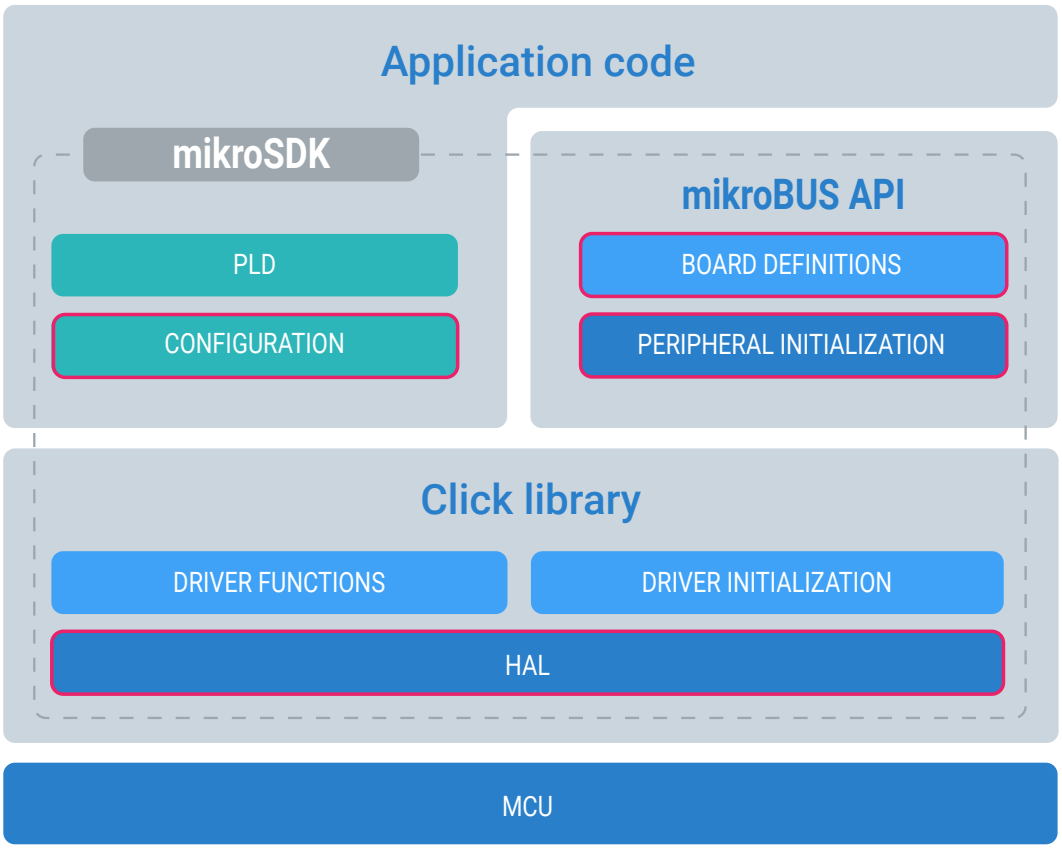


Figure 2: mikroSDK, separated in 3 main elements: click library, mikroBUS API and application code

This way, most of the layer code that is identical for all the click boards™ that use the same peripheral, can be reused. Also, the application code and the Driver code is identical for all the supported architectures and it is independent of the underlying architecture on which the code is being executed. This code modularity, allows specific layers to be swapped according to needs so that the specific layer code does not have to be redeveloped every time.

mikroBUS API and the HAL have fully defined interfaces, required to be implemented for the successful porting, which leads to the possibility to have easy-to-use templates as the starting point for extending the mikroSDK with the new architectures and platforms. Compared to that, the Driver layer should be able to be compiled on every ANSI C compliant compiler.

Further analysis will be done by dividing the mikroSDK to its elements like shown in Figure 2: the click library, the mikroBUS API layer, and the application code. Each of

these elements will be analyzed separately since every one of these parts represents an integral part of the mikroSDK.

NOTE

For a practical application example of the mikroSDK, you can skip directly to the **Practical usage of the mikroSDK in mikroE compilers.**

SUMMARY

The mikroSDK standard defines specific layers, categorized by the physical domain they are written for: MCU architecture specific code is placed in a separate layer, development platform specific code is placed in its own separate layer, and finally - the application software with the configuration parameters is also placed in a separate layer. This modular approach allows the components to be switched, along with the corresponding software layers.

CLICK LIBRARY

Introduction

Standardization of the Click library format covers the library implementation and code organization; it also defines the rules to follow, to make the Driver mikroSDK standard compliant. The standard allows the existing click library to be easily expanded with the support for new architectures.

The Click library is composed of two layers: The Driver layer and the Hardware Abstraction Layer, or shorter - the HAL.

The Hardware Abstraction Layer (HAL) is the lower-level layer with the common interface on all supported architectures/compilers. This layer has no function accessible from the user-space.

The Driver layer (Driver) is the reusable upper-level layer, which contains all the available user functions. It also contains the additional content, such as the constants for the register addresses and other constants available for the example code, which can be used along with the available functions. Some of the content is not necessary but makes implementation of the final application code much easier and much more comfortable.

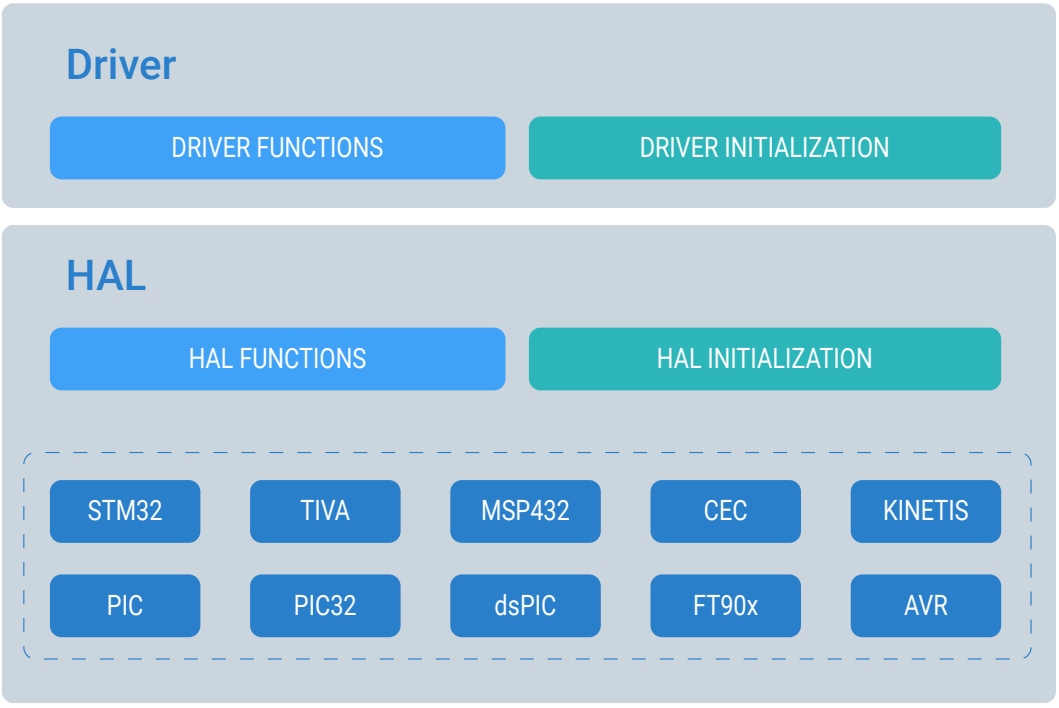


Figure 3: Two main sections of the mikroSDK standard compliant click board™ library

mikroSDK standard strictly prescribes that each library has a unique prefix, named mostly related to the click board™ name, which ensures that there are no conflicting global names from two different libraries. Prefix usage is mandatory for all globally accessible identifiers.

Libraries that are provided for the MikroElektronika compilers are distributed as the .mpkg packages, which contain object files, additional help documentation, and the demonstration application examples. However, all the libraries will also be available in a form of an open source for the mikroC compilers.

NOTE

Libraries, developed according to mikroSDK standard, can be used in other MikroElektronika programming languages - mikroBasic and mikroPascal. Before the libraries can be used, they have to be compiled for each architecture / MCU vendor, by using the mikroC compiler and by generating MikroElektronika object files ***(.mcl and .emcl)***.

SUMMARY

The click board libraries are provided as the software support for the click boards™. The standardization of the click libraries allows for the click board application to run on any supported hardware platform, without the need to rewrite the code itself. The click library contains two layers – The HAL layer and the Driver layer.

HAL (Hardware Abstraction Layer)

The purpose of the Hardware Abstraction Layer (HAL) is to overcome the differences between the used architectures, both in software and hardware terms. This is the layer that allows for the library to be ported among other architectures/compilers.

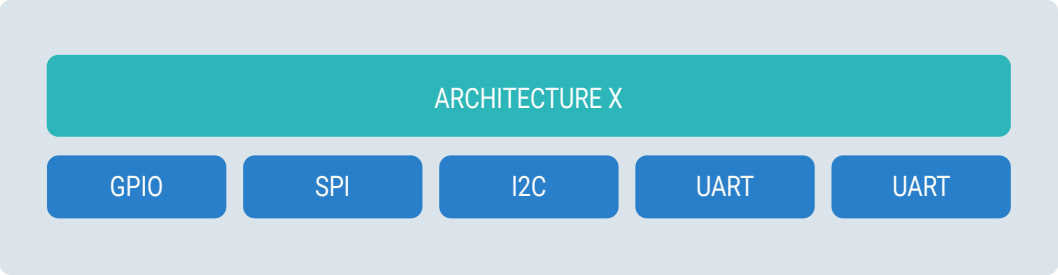


Figure 4: Hardware Abstraction Layer (HAL) and the supported peripherals

The mikroC HAL is composed of the source code files - one for each MCU architecture, where one file covers all the peripherals currently supported by the standard *(Figure 5)*.

Each supported peripheral has functions derived from the unique interface, which makes the peripherals look identical on all the supported MCU architectures; the HAL abstracts the peripheral functions. This property of the HAL simplifies the development of the upper (Driver) layer and makes it independent of the used architecture.

Figure 5: Different architectures and their peripheral modules

NOTE

Once written, HAL layer for any compiler can be re-used for any future projects.

SUMMARY

The HAL is used to abstract the peripherals functions so that the functions look the same on all the supported MCU architectures. This property of the HAL greatly simplifies the Driver development. Once written, the HAL can be re-used for any future projects.

Compiling of the HAL

The mikroC HAL is composed of modules written for different architectures. During the compiling of the HAL for the specific architecture, the presence of other architecture related content must not be allowed. Because of this, the mikroSDK standard prescribes using of the HAL selectors. The HAL selectors are nothing more than preprocessors, which do not allow compiling of parts of the HAL that are not related to the architecture or peripheral used by the Driver layer.

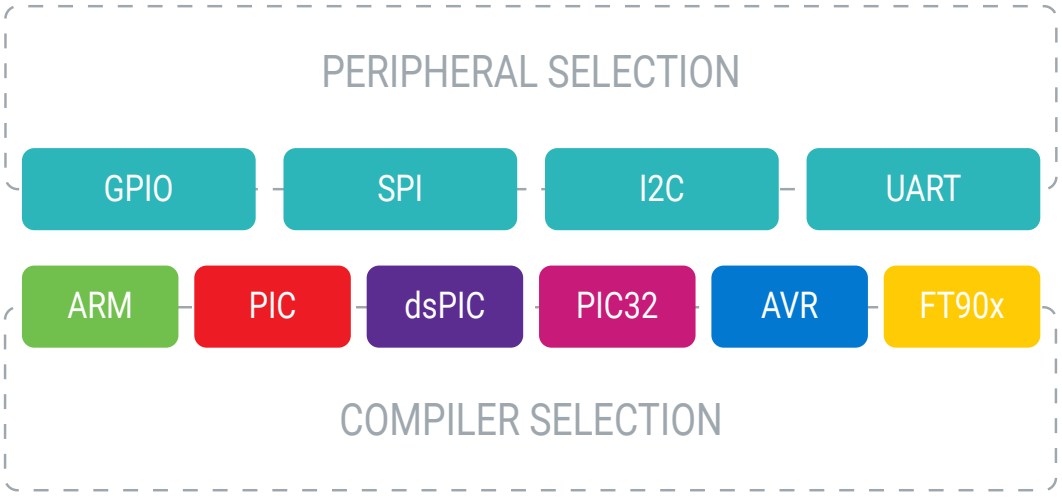


Figure 6: HAL compiling selectors

Architecture selectors

Architecture and vendor selectors are provided automatically by the compiler command line.

```
#define __MIKROC_PRO_FOR_ARM__
#define __MIKROC_PRO_FOR_AVR__
#define __MIKROC_PRO_FOR_PIC__
#define __MIKROC_PRO_FOR_DSPIC__
#define __MIKROC_PRO_FOR_PIC32__
#define __MIKROC_PRO_FOR_FT90x__
```

ARM vendor selectors

```
#define __STM32__
#define __TI__
#define __MSP__
#define __MCHP__
#define __KINETIS__
```

Peripheral selectors

Peripheral selectors are defined at the top of the library and used through the whole source file.

```
#define __CLICK_GPIO
#define __CLICK_SPI
#define __CLICK_I2C
#define __CLICK_UART
```

SUMMARY

When the HAL is compiled, only the content for the used peripheral and architecture should be included, leaving out the code written for other architectures or peripherals. That is why the standard defines HAL selectors, preprocessors which allow only relevant content to be compiled.

HAL implementation

HAL is implemented so that it exposes the identical interface for any architecture and is composed of identical function calls, internally linked by the Driver layer. Implementation for each supported architecture is placed inside the separate source file.

NOTE

Implementation must follow the Coding rules (Chapter 5)

During the initialization, pointers to the toolchain specific functions are provided from the higher layer and stored inside the HAL as static members. These pointers are wrapped inside the HAL functions, allowing the upper layer to use the same function calls on all MikroElektronika compilers, independently of the underlying architecture - as long as it is supported by the MikroElektronika compilers.

HAL types

Each part of the HAL has its own structures, composed of function pointers used internally for communication with the hardware. These structures are used only during the initialization to assign proper values to the static function pointers.
Pointer prototypes depend on the architecture, except for the GPIO function pointer types, which are identical for all the MCU architectures.

Since prototypes of peripheral functions differ from one compiler to another, HAL layer takes care of bridging those differences. This property of HAL allows for easy porting of the HAL to a newly supported architecture.

GPIO types

GPIO function pointer types are identical across all the mikroC compilers for different platforms. GPIO structure contains function pointers for each GPIO pin found on the mikroBUS™, starting from the AN pin (Index 0), up to SDA pin (Index 11).

```
typedef void    (*T_click_gpio_Set)(uint8_t);
typedef uint8_t (*T_click_gpio_Get)();
typedef struct
{
   T_click_gpio_Set      gpioSet[ 12 ];
   T_click_gpio_Get      gpioGet[ 12 ];

}T_click_gpioObj;
```

SPI types

Read and write functions are mostly the same on all mikroC compilers. The only differences are related to the argument and return value types. While SPI read function pointer alone is sufficient for the SPI communication in both directions, both SPI read and write pointers are defined inside the SPI structure, for sustained backward compatibility.

```
typedef struct
{
    T_click_spi_Write       spiWrite;
    T_click_spi_Read        spiRead;

}T_click_spiObj;
```

I2C types

Implementation of the I2C libraries varies the most across the different mikroC compilers. Not every mikroC compiler has all the function pointer types located inside the I2C structure.

```
typedef struct
{
    T_click_i2c_Start       i2cStart;
    T_click_i2c_Stop        i2cStop;
    T_click_i2c_Restart     i2cRestart;
    T_click_i2c_Write       i2cWrite;
    T_click_i2c_Read        i2cRead;

}T_click_i2cObj;
```

UART types

Similar to the SPI peripheral, differences between UART function pointer types are mostly related to the arguments and return value types.

```
typedef struct
{
    T_click_uart_Write      uartWrite;
    T_click_uart_Read       uartRead;
    T_click_uart_Ready      uartReady;

}T_click_uartObj;
```

SUMMARY

The HAL uses its own structures to communicate with the hardware. Those structures are only used during the initialization, to assign proper values to the static function pointers.

HAL initialization functions

All the HAL Init functions are static and internally linked by the Driver layer.

The initialization within the Driver layer connects the HAL functions and proper peripheral function calls (such as writing and reading functions for SPI, I2C, UART, etc.).

The Init functions for a certain peripheral are named by using the Map suffix and always have the same prototype, independent of the used compiler. The function argument is always abstract type, defined inside the Driver layer, using the same macro for all the mikroC compilers:

```
#define T_HAL_P const uint8_t*
```

Each HAL public function expects a pointer to the appropriate structure, cast to the HAL abstract type. These structures contain function pointers for that specific peripheral and those pointers will be assigned to the private function pointer members, inside the HAL.

Initialization function for the HAL subsection of each peripheral must be called first, before using any other Driver function that uses functions from that HAL subsection. The provided argument will be dereferenced to the appropriate HAL Types.

GPIO map

```
void hal_gpioMap(T_HAL_P gpioObj);
```

Initialization of the GPIO HAL

Parameters:

gpioObj	pointer to the GPIO structure which carries proper pointers

SPI map

```
void hal_spiMap(T_HAL_P spiObj);
```

Initialization of the SPI HAL

Parameters:

spiObj	pointer to the UART structure which carries proper pointers

I2C map

```
void hal_i2cMap(T_HAL_P i2cObj);
```

Initialization of the I2C HAL

Parameters:

i2cObj	pointer to the I2C structure which carries proper pointers

UART map

```
void hal_uartMap(T_HAL_P uartObj);
```

Initialization of the UART HAL

Parameters:

uartObj	pointer to the UART structure which carries proper pointers

NOTE

HAL initialization does not execute peripheral initialization. Peripheral initialization must be executed explicitly inside the user application or by the mikroBUS API.

SUMMARY

The HAL Init functions are called from within the Driver layer and are used to map the HAL function pointers to the peripheral function calls. The HAL initialization has to be done before any other HAL functions of that subsection are accessed from the Driver layer. Since the HAL Init function only maps the appropriate pointers, no peripheral initialization is going to be executed by this function.

HAL functions

All HAL functions are internally linked by the Driver layer and have the same prototype on all compilers.

Each peripheral section has its own set of functions so that the set of functions can be observed as a separate module. Implementation of these peripheral related functions is the key to successful porting of the HAL.

These functions must use only function pointers provided during the initialization. This allows for a dynamic assignment and usage of the same library for different physical modules, provided that the higher layers are properly written.

GPIO

HAL functions related to the GPIO are nothing more than function pointers named with the hal_ prefix. Since the HAL compiling unit is internally linked with the Driver layer, the Driver is able to call execution of these pointers, directly.

Compared to the other HAL modules, compiling of the GPIO module functions depends on additional selectors, defined at the top of the HAL interface. Each GPIO pin has two selectors: the first one is for the input direction and the second one for the output direction.

```
#define __AN_PIN_INPUT__
#define __RST_PIN_INPUT__
#define __CS_PIN_INPUT__
#define __SCK_PIN_INPUT__
#define __MISO_PIN_INPUT__
#define __MOSI_PIN_INPUT__
#define __PWM_PIN_INPUT__
#define __INT_PIN_INPUT__
#define __RX_PIN_INPUT__
#define __TX_PIN_INPUT__
#define __SCL_PIN_INPUT__
#define __SDA_PIN_INPUT__

#define __AN_PIN_OUTPUT__
#define __RST_PIN_OUTPUT__
#define __CS_PIN_OUTPUT__
#define __SCK_PIN_OUTPUT__
#define __MISO_PIN_OUTPUT__
#define __MOSI_PIN_OUTPUT__
#define __PWM_PIN_OUTPUT__
#define __INT_PIN_OUTPUT__
#define __RX_PIN_OUTPUT__
#define __TX_PIN_OUTPUT__
#define __SCL_PIN_OUTPUT__
#define __SDA_PIN_OUTPUT__
```

Commenting out the unneeded pin definitions will remove the specific pin function pointers from the HAL. Using this method will allow having only the necessary pointers in the code, saving the RAM memory that way and optimizing the HAL.

SPI

SPI functions are wrappers for the function pointers provided during initialization, with the addition of sequence read and write, because of the sequence execution speed improvements.

```
static void hal_spiWrite
(uint8_t *pBuf, uint16_t nBytes);
```

Function should execute write sequence of n bytes.

Parameters:

out	pBuf	pointer to data buffer
in	nBytes	number of bytes for writing

```
static void hal_spiRead
(uint8_t *pBuf, uint16_t nBytes);
```

Function should execute read sequence of n bytes.

Parameters:

out	pBuf	pointer to data buffer
in	nBytes	number of bytes for reading

```
static void hal_spiTransfer
(uint8_t *pIn, uint8_t *pOut, uint16_t nBytes);
```

Function should execute RW sequence of n bytes

Parameters:

in	pIn	pointer to write data buffer
out	pOut	pointer to read data buffer
in	nBytes	number of bytes for exchange

I2C

I2C static functions are modeled by the functions used for the STM32, found in the mikroC PRO for ARM compiler. The main reason for this is to simplify the development of the library. Code, written for the STM32 can be ported to the library by simply renaming the compiler I2C functions to the HAL equivalent ones.

```
static int hal_i2cStart();
```

This function in the snippet above should execute start condition on the I2C BUS.

```
static int hal_i2cWrite
(uint8_t slaveAddr, uint8_t *pBuff, uint16_t nBytes, uint8_t endMode);
```

This function in the snippet above should execute write sequence, write the data inside the pBuf and execute “end” or “restart” condition, depending on the endMode argument.

Parameters:

in	slaveAddress	7 bit slave addres without 0 bit (read/write bit)
in	pBuf	pointer to data buffer
in	nBytes	number of bytes for writing
in	endMode	END_MODE_STOP / END_MODE_RESTART

```
static int hal_i2cRead
(uint8_t slaveAddr, uint8_t *pBuff, uint16_t nBytes, uint8_t endMode);
```

This function in the snippet above should execute read sequence, and place the data inside the pBuf and execute "end" or "restart" condition, depending on the endMode argument.

Parameters:

in	slaveAddress	7 bit slave addres without 0 bit (read/write bit)
out	pBuf	pointer to data buffer
in	nBytes	number of bytes to read
in	endMode	END_MODE_STOP / END_MODE_RESTART

UART

UART static functions are wrappers for the familiar read, write and ready functions, with common usage on all compilers.

```
static void hal_uartWrite(uint8_t input);
static uint8_t hal_uartRead();
static uint8_t hal_uartReady();
```

SUMMARY

The HAL functions have the same prototype on all compilers, which is the ultimate goal of the HAL layer. This makes the upper layer able to use the same functions for the specific peripheral module.

Driver layer

The Driver layer contains the main implementation of the click board™ functionality. This layer must be written in pure ANSI C89 without using of any mikroC specific functions, macros or any of mikroC libraries, as for example - the conversion functions, because this layer must be able to compile in any ANSI C compliant compiler.

This allows for the extension of the HAL layer at a later time and using the Driver with other compilers. Driver access to hardware is allowed only through the HAL functions calls.

Driver implementation

Implementation must follow the Coding rules (Chapter 5).
The standard prescribes the content of driver files very strictly. The Driver files content can be divided into sections, as shown in the *Figure 7*.

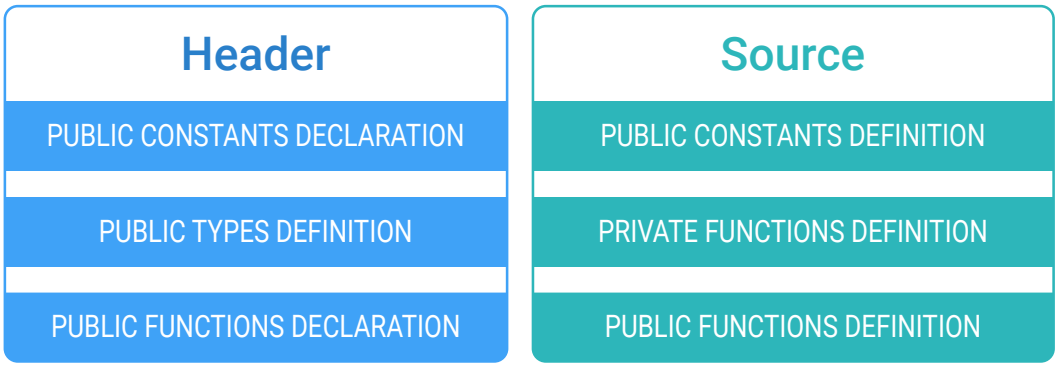


Figure 7: Structure of the driver layer

Driver initialization

Driver initialization functions are the most direct interaction between the user space and the HAL because the Driver initialization functions are mostly wrappers for the HAL initialization functions.

Driver initialization functions are the only common functions for all the Drivers which use the same peripheral.

The initialization function arguments are common abstract types, which expect a pointer to structures, with the proper toolchain function pointers as their structure members. These pointers are forwarded directly to the HAL initializations and stored inside the HAL.

Abstract type used for all initialization functions, is defined by using a macro and contains prefix name to avoid any collision with the same type, defined inside some other library:

```
#define T_CLICK_P const uint8_t*
```

Driver initialization functions can be observed in the following code snippet:

```
void CLICK_spiDriverInit
(T_CLICK_P gpioObj, T_CLICK_P spiObj);

void CLICK_i2cDriverInit
(T_CLICK_P gpioObj, T_CLICK_P i2cObj, uint8_t slaveAddr);
void CLICK_uartDriverInit
(T_CLICK_P gpioObj, T_CLICK_P uartObj)

void CLICK_gpioDriverInit
(T_CLICK_P gpioObj);
```

Each Driver initialization function has a pointer to the GPIO structure as the first argument. The second argument is the pointer to the HAL structure, which contains function pointers for the toolchain specific peripheral functions. Also, additional arguments are allowed like in the case of the I2C Driver, where providing the slave address is mandatory for the Driver.

Driver constants

Driver constant variables should be declared as external constants inside the header and defined inside the C file. Those variables are mostly related to the values such as register addresses or some specific settings, or even function return values.

H file

```
extern const uint8_t _CLICK_SOME_VAR;
extern const uint8_t _CLICK_SOME_ARR[];
```

C file

```
const uint8_t _CLICK_SOME_VAR = 15;
const uint8_t _CLICK_SOME_ARR[3] = { 10, 20 ,30 };
```

By using constants, it is ensured that the library can be used either as an object file or as a source code. This is important when usage of the same object files for mikroPascal and mikroBasic is required.

SUMMARY

The Driver layer is directly interfaced with the HAL layer. The Driver initialization functions are actually wrappers of the HAL initializations. The first argument of the Driver Init functions is always a pointer to the GPIO structure, followed by the argument that is the pointer to the HAL structure which contains function pointers for the toolchain specific peripheral functions (such as the SPI, I2C...). The Driver layer has to be ANSI C89 compliant so that the portability can be retained.

Library documentation

Documentation for the each part of the mikroSDK is provided in a form of .chm - windows help files, placed inside the **.mpkg** package.

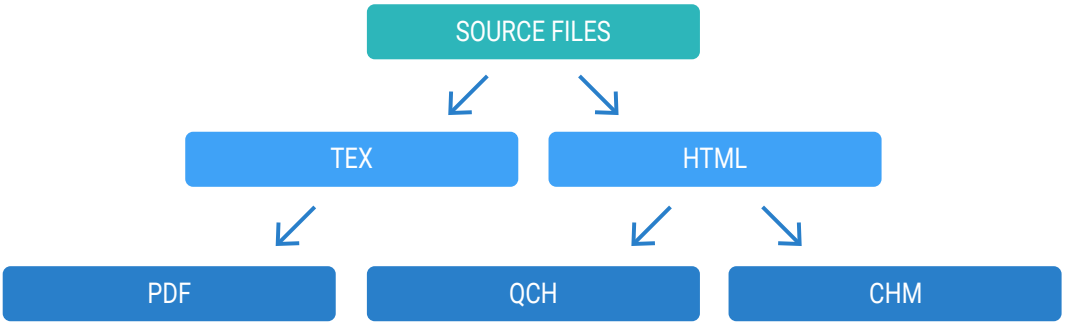


Figure 8: Schematic diagram of Help file generation process

Generating of the Help can be illustrated graphically, like in the table above *(Figure 8)*.

Generating HTML by using Doxygen is essentially a transition process. This process is used for customization of the final documentation, by using the .css files, provided by MikroElektronika.

Generating the .chm file is a conversion of the generated HTML documentation to .chm format (Windows OS help format) with no additional operation or text formatting needed.

For generating the documentation, a template is already provided. It must be used to generate documentation for each click board™. This way, provided documentation will be uniformly formatted – it will have the same style and format for all the click boards™.

More details about the documentation rules can be found in the documentation rules section.

SUMMARY

The mikroSDK standard also defines the structure of the documentation. The software used for the documentation generation is the Doxygen, a widely accepted software tool which generates uniform and clean technical documentation for the developed software.

mikroBUS API

Introduction

The mikroBUS API is a distinct layer between the user application and the click Driver, which represents an abstraction of all supported development systems, focused on systems equipped with the mikroBUS™ sockets.

The main advantage of using the mikroBUS API is that it overcomes dissimilarities between the different development systems, allowing the developer to use exactly the same code for applications on all the supported architectures. The mikroBUS API consists of board definition files, one for each supported development system. Board definition files contain definitions for each mikroBUS™ present on the specific system, as well as some additional modules, or even single helper functions.

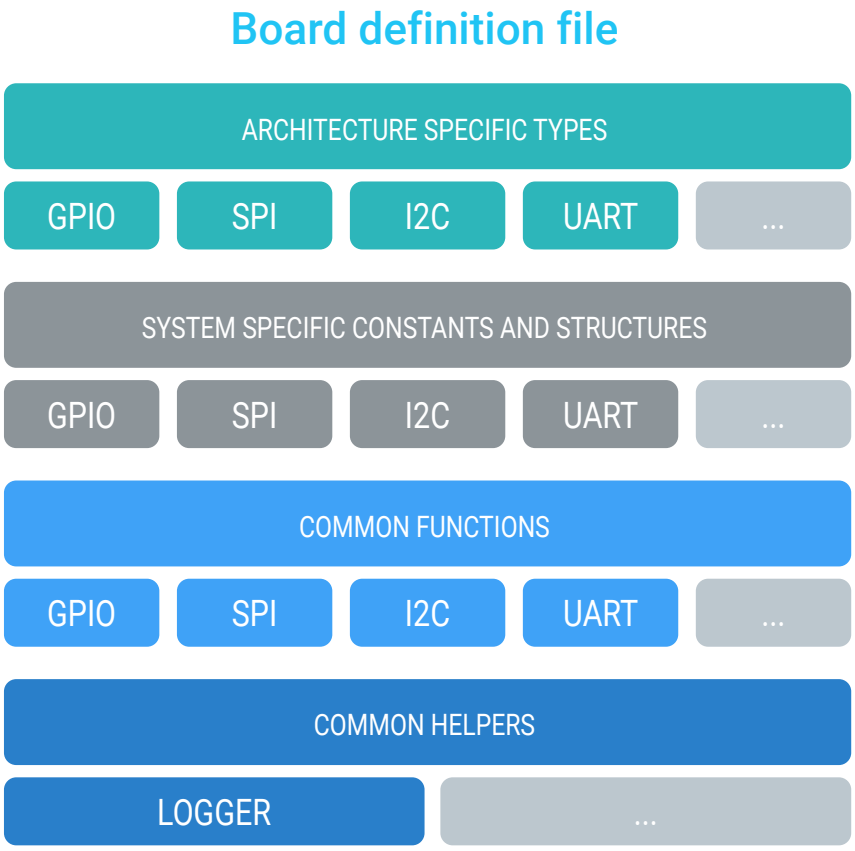


Figure 9: Board definition file structure

While HAL simplifies and makes library identical for all MCU architectures/vendors, The mikroBUS API simplifies and makes the provided demo applications and user

applications seem identical on all the supported hardware platforms, allowing them to use the same simple API calls for any development system. These API calls are mostly related to Driver initialization, except for some additional functions.

Board definition files are implemented using the identical interface that is placed at the top level, while modules divided by the peripherals currently supported by the mikroSDK are located in a level below this interface.

NOTE

If this standard is followed during the implementation of every sublayer, example provided will be the same for all platforms supported by mikroSDK.

SUMMARY

The mikroBUS API represents an abstraction layer for the actual hardware that runs the application. The mikroBUS API consists of board definition files which contain definitions for each mikroBUS™ present on the specific system. The mikroBUS API makes the application code seems identical for all the supported hardware platforms, regardless of the physical differences among them. Extension of the mikroBUS API is possible in 3 directions: by adding the board definition files to support additional systems, by adding new supported peripherals, or by adding various helper functions or modules.

mikroBUS API compiling

Compiling of the mikroBUS API content mostly depends on the content of the PLD file and the selected board in the Library Manager. PLD files are composed of selectors used to decide which part of the board definition files are going to be compiled, conserving the memory footprint of the code that way. PLD file can be replaced with a simple header, included in every interface and module inside the mikroBUS API.

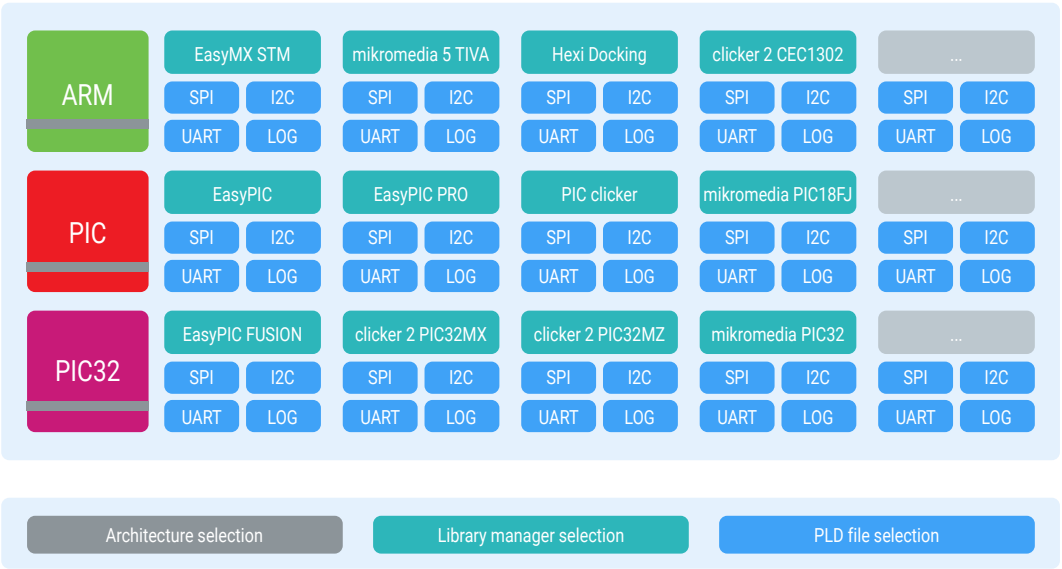


Figure 10: mikroSDK compiling selectors

Similarly, like the sections of the HAL layer, mikroBUS API board definition files can also be divided into four main sections: GPIO, SPI, I2C, and UART. Compiling of each section depends on the content of the PLD file, except for the GPIO part - which is always compiled.

Depending on the MikroElektronika Library Manager board selection, only one board definition file will be compiled. Peripheral selection in PLD file is used to determine which modules and interfaces of the selected board definition file will be compiled (SPI, I2C, UART). This can optimize the memory usage. Figure 10 illustrates all the factors which decide which part of the mikroBUS API will be compiled.

Peripheral selectors

```
ENABLE_SPI
ENABLE_I2C
ENABLE_UART
```

Helper selector

```
ENABLE_LOG
```

SUMMARY

mikroBUS API is also subject to selective compiling, just like the HAL. The selectors for the mikroBUS API sections come in a form of PLD file directives and the Library Manager board selection. Leaving out parts of the mikroBUS API that are not needed results with the smaller application footprint.

mikroBUS API implementation

mikroBUS API modules

Each mikroBUS API module is composed of structures and additional functions which are used for peripheral initialization.

mikroBUS API must be provided in a form of an open source code. Source code is placed inside the **.mpkg** package, for each program language and MCU architecture.

Structure definitions are identical to structures defined inside the HAL and the members (function pointers) prototypes depend on the used architecture. Functions which perform peripheral initialization, always have the same prototype and a single argument, which contains parameters for peripheral configuration.

GPIO module

GPIO module contains GPIO functions for setting and getting the state of each individual pin of the mikroBUS™ slots. These functions are stored inside the structures which are passed to the HAL layer during the Driver initialization. Board def also contains GPIO set direction functions for each individual pin, so it can be set as either input or output at any time.

Peripherals modules

There are multiple sets of SPI, UART and I2C functions defined for each system in the compiler libraries. Board definition files take care of selecting the correct functions for the desired mikroBUS™ of the chosen development system. Pointers to those functions are stored inside the mikroBUS API structures and will be passed to the HAL layer during the Driver Init function call.

Board definition files also contain initialization functions for each peripheral.

Logger module

The logger uses built-in UART functions to send data from the user application to the selected UART output. It is configured and initialized using the provided function, from

inside the user application. Logger is not limited to sending data via the mikroBUS™ UART pins only. It can also send data via USB UART ports, on development systems where those are included.

Logger is the only part of the mikroSDK that has no connection to the Driver.

mikroBUS API Types

From the developer's point of view, the most important mikroBUS API types are structures used for the library initialization.

Every board def must have at least one public variable of the appropriate type per each mikroBUS™. These variables contain pointers to MikroElektronika library functions, related to the specific mikroBUS™ and peripheral module. These variables are provided to library Driver initialization function.

GPIO types

Structure identical to `T_hal_gpioObj .`

```
typedef struct
{
    T_gpio_set gpioSet[ 12 ];
    T_gpio_get gpioGet[ 12 ];

}T_gpio_obj;
```

NOTE

There are 12 set members and 12 get members according to the amount of available GPIO pins on the mikroBUS.

SPI types

Structure identical to `T_hal_spioObj .`

```
typedef struct
{
    T_spi_write  spiWrite;
    T_spi_read   spiRead;

}T_spi_obj;
```

I2C Types

Structure identical to `T_hal_i2cObj .`

```
typedef struct
{
    T_i2c_start     i2cStart;
    T_i2c_stop      i2cStop;
    T_i2c_restart   i2cRestart;
    T_i2c_write     i2cWrite;
    T_i2c_read      i2cRead;

}T_i2c_obj;
```

UART types

Structure identical to `T_hal_uartObj .`

```
typedef struct
{
    T_uart_write    uartWrite;
    T_uart_read     uartRead;
    T_uart_ready    uartReady;

}T_uart_obj;
```

mikroBUS API public functions

Each board def module has at least one public function per peripheral, used for peripheral initialization. All of the functions contain the mikroBUS™ slot as the first argument. The rest of the arguments are used to configure other peripheral specific parameters.

GPIO Functions

```
T_mikrobus_ret mikrobus_gpioInit
(T_mikrobus_soc bus, T_mikrobus_pin pin, T_gpio_dir direction);
```

The function in the code snippet above configures the GPIO pin, according to the provided parameters. This function must be called once per each GPIO used by the Driver.

Parameters:

in	bus	mikroBUS™ number
in	pin	mikroBUS™ pin
in	direction	GPIO direction

Returns:
`_MIKROBUS_OK / _MIKROBUS_ERR_BUS / _MIKROBUS_ERR_PIN`

```
T_mikrobus_ret ret = 0;

ret |= mikrobus_gpioInit( _MIKROBUS1, _MIKROBUS_INT_PIN, _GPIO_INPUT );
ret |= mikrobus_gpioInit( _MIKROBUS2, _MIKROBUS_CS_PIN, _GPIO_OUTPUT );

if (0 == ret)
{
   // SUCCESS
}
else
{
    // ERROR
}
```

This example demonstrates a proper way for GPIO direction setup. INT pin on mikroBUS™ 1 is configured as input and CS pin on the second mikroBUS™ is configured as an output.

SPI functions

```
T_mikrobus_ret mikrobus_spiInit
(T_mikrobus_bus bus, const uint32_t *cfg);
```

The function will initialize SPI peripheral on the mikroBUS™ provided as the first argument, with the configuration provided as the second argument. Default peripheral configuration is provided along with the click board™ library.

Parameters:

in	bus	bus number
in	cfg	pointer to the I2C configuration

Returns:
`MIKROBUS_OK / MIKROBUS_ERR_BUS / MIKROBUS_ERR_I2C`

UART functions

```
T_mikrobus_ret mikrobus_uartInit
(T_mikrobus_bus bus, const uint32_t *cfg);
```

The function in the snippet above will initialize UART peripheral on the mikroBUS™ provided as the first argument, with the configuration provided as the second argument. Default peripheral configuration is provided alongside with the click board™ library.

Parameters:

in	bus	bus number
in	cfg	pointer to the UART configuration

Returns:
`MIKROBUS_OK / MIKROBUS_ERR_BUS / MIKROBUS_ERR_UART`

LOG functions

LOG module compared to other modules have two public functions. One is for the initialization and the other one is for the data logging.

```
T_mikrobus_ret log_init
( T_log_port port, const uint32_t baud );

T_mikrobus_ret log_write
( uint8_t* data, T_LOG_format format );
```

LOG module requires initialization before using its log write function. Init function will initialize proper UART module, depending on the first argument and selected development system with baud rate provided as the second argument, with 8 data bits, no parity and one stop bit (8N1).

SUMMARY

The mikroBUS API layer consists of structure definitions, identical to the structures defined in the HAL. The board definition files are used to select the specific peripheral, routed to the desired mikroBUS™. Pointers to those peripheral functions are stored inside the mikroBUS API structures and are passed to the HAL layer during the Driver Init function calls. The mikroBUS API public functions are used to initialize the actual hardware platform that is used to run the application.

Demo application code

Introduction

Demo application source code, as the most exposed part of this standard, should be simple and well organized. Provided demo applications will be usable in the "out of box" manner for the default MikroElektronika development systems.

Example contains following files:

1. Source file **.c, .mpas, .mbas**
2. Project file .**mcp, .mpp, .mbp**
3. Project configuration file **.cfg**
4. Default peripherals configuration for the particular click board™ provided in file **.h, .mpas, .mbas**
5. Additional necessary types **.h, .mpas, .mbas**
6. PLD File **.pld**

Source file

The source file contains the source code of the demo application. It includes the demo application description and details contained in the comments and implementation of the example.

Project file

The project file contains the project settings for the MikroElektronika compiler projects. It is generated automatically, and should generally be left unchanged.

Project configuration file

Project configuration contains the configuration bits settings for the selected MCU. It is changed by loading schemes in the Edit Project window of MikroElektronika compilers.

Peripherals configuration file

This header file contains constants used for the initialization of the specific peripheral.

These constants represent the default peripheral initialization parameters for a specific click board™, which is provided to the mikroSDK initialization function, for the specific module.

```
const uint32_t _CLICK_SPI_CFG[ 3 ] =
{
    1000000,
    _SPI_MSB_FIRST,
    _SPI_CLK_IDLE_LOW |
    _SPI_SAMPLE_DATA_RISING_EDGE
};
```

Content of these constants depends on compiler/architecture and vendor.

Additional types

This header file contains necessary types, such as T_CLICK_P abstract type used for casting during Driver initialization calls.

PLD file

PLD file contains peripheral selectors, generated automatically by the automatization application, or placed inside manually, by the end-user.

Demo application code implementation

Usage of the **mikroBUS API** LOG helper functions is preferred.
Each demo application code should be composed of three basic functions whenever it is possible:

1. **systemInit**
2. **applicationInit**
3. **applicationTask**

`systemInit` is the place for mikroBUS API initialization function calls.
`applicationInit` function is the right place for the driver initialization function and other necessary calls, related to the library initialization and calls needed to be done before using the driver functionalities.
`applicationTask` function will be placed inside an infinite loop, and it carries a simple demonstration of the driver functions usage.

Application example

The pseudo-example below demonstrates how to use the mikroSDK in practice. Provided example could be used on any architecture/mikroC compiler, because all the code related to some specific MCU architecture is now placed inside the **mikroSDK**. This simple example uses SPI, GPIO, and LOG.

PLD file content

```
ENABLE_SPI
ENABLE_LOG
```

mikroC example code

```
// Not needed in case of using of MikroElektronika packages
#include "__click_driver.h"

// Default config provided with click board for each arch
#include "Click_config.h"

#include "Click_types.h"

// COMMON TYPES - default types
void systemInit()
{
    mikrobus_gpioInit( _MIKROBUS1, _MIKROBUS_CS_PIN, _GPIO_OUTPUT );
    mikrobus_spiInit( _MIKROBUS1, _CLICK_CFG );
    mikrobus_logInit( _MIKROBUS2, 9600 );
    Delay_ms( 100 );
}

void applicationInit()
{
     click_driverInit( (T_CLICK_P)&_MIKROBUS1_GPIO, (T_CLICK_P)&_MIKROBUS1_SPI );
     mikrobus_logWrite( "Initialized", _LOG_LINE );
}

void applicationTask()
{
    char readValue;
    char testTxt[ 25 ];

    click_writeRegister( _CLICK_REG, 0xD0 );
    readValue = click_readRead( _CLICK_REG );

    ByteToStr( readValue, testTxt );                // Conversion
    mikrobus_logWrite( "Value : ", _LOG_TEXT );     // Writing text to UART
    mikrobus_logWrite( testTxt, _LOG_LINE );        // Writing register content
    Delay_ms(1000);
}
void main()
{
    systemInit();
    applicationInit();
    while (1)
    {
        applicationTask();
    }
}
```

It is also possible to avoid any use of the mikroSDK. In that case, it is up to end user to send the correct function pointers to the HAL layer. Peripheral functions are already available in MikroElektronika compilers, but the GPIO set/get functions need to be implemented manually.

```
#include "__click_driver.h"

void csControl( uint8_t state )
{
    GPIOD_ODR.B13 = state;
}

// Structure which holds pointer to function which set the state of
CS pin
const T_hal_gpioObj myGpioMap =
{
    { 0, 0, csControl, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
};

// Structure which holds pointers to MikroElektronika SPI library
functions // // used on desired bus
const T_hal_spiObj mySpiMap =
{
    SPI3_Write,
    SPI3_Read
};

void systemInit()
{
    GPIO_Digital_Output( &GPIOD_ODR, _GPIO_PINMASK_13 );
    SPI3_Init_Advanced( _SPI_FPCLK_DIV256, _SPI_MASTER | _SPI_8_BIT
|
                        _SPI_CLK_IDLE_HIGH | _SPI_SECOND_CLK_EDGE_
TRANSITION |
                        _SPI_MSB_FIRST | _SPI_SS_DISABLE | _SPI_SSM_
ENABLE |
                        _SPI_SSI_1, &_GPIO_MODULE_SPI3_PC10_11_12 );
    UART1_Init( 9600 );
    Delay_ms( 100 );
}
void applicationInit()
{
    click_driverInit( (const uint8_t*)myGpioMap, (const uint8_t*)
mySpiMap );
    UART1_Write_Text( "Initialized\r\n" );
}

void applicationTask()
{
    char readValue;
    char testTxt[25];

    click_writeRegister(_CLICK_REG, 0xD0);
    readValue = click_readRegister(_CLICK_REG);
    ByteToStr(readValue, testTxt);
    UART1_Write_Text("Value: " );
    UART1_Write_Text( testTxt );
    UART1_Write_Text( "\r\n" );
    Delay_ms(1000);
}

void main()
{
    systemInit();
    applicationInit();

    while (1)
        applicationTask();
}
```

These two examples above can be used to achieve the same thing. The main difference is that the first code where the mikroSDK is used is smaller and way easier to read. It is not dependent on the underlying architecture. The second code is made to run strictly on the EasyMX for STM32 board and the specific MCU, since all the GPIOs and compiler built-in functions are specifically initialized in the code.

SUMMARY

The mikroSDK Standard clearly defines the existence of the demo application, which demonstrates the basic functionality of the target device for which the application is developed, in an out-of-the-box manner. The demo application has to follow the rules defined by the mikroSDK standard and as such, it can be used as a starting point or a reference for the future design of the user's application.

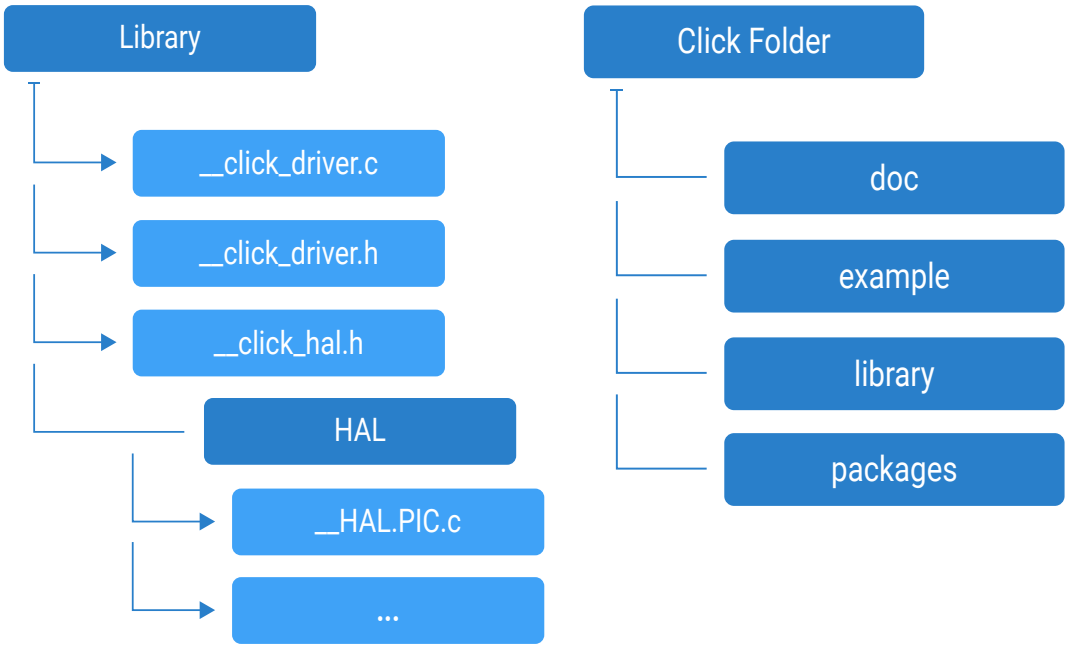
Coding rules

Introduction

Unlike some architecture, dependent parts of the mikroSDK, like the board definition files and the HAL, which are developed for some specific compiler, the Driver layer is the only one which must be able to compile, regardless of the used compiler. Because of that, most of the coding rules are related to the Driver layer.

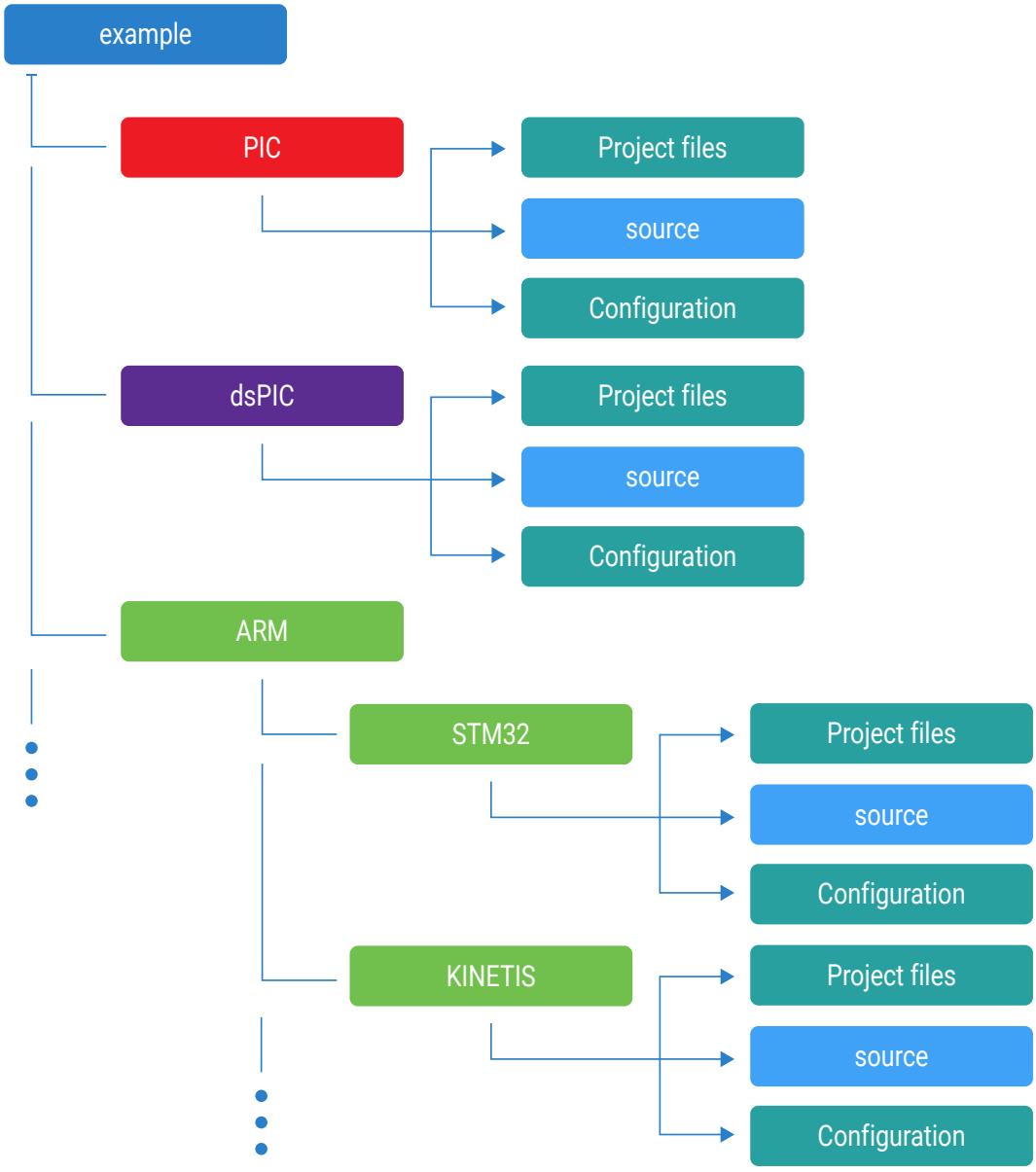
Files and folders organization

This set of rules also covers folder and files organization and documentation provided with the click board™.



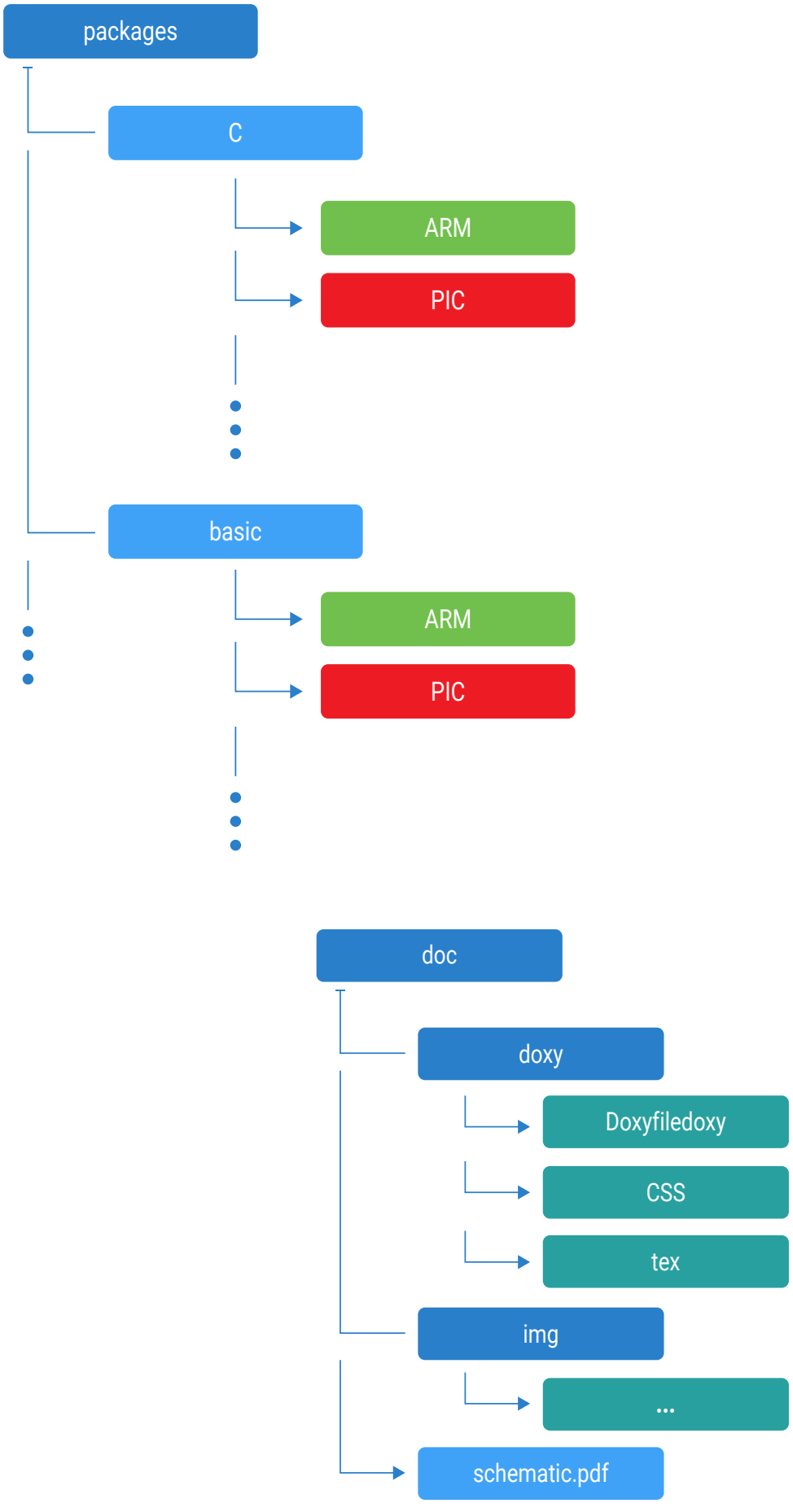
Click library file names always start with the "__" double underscore character. All content related to the click library is placed inside the library folder. While HAL interface itself is placed inside the library, all the things related to some specific architecture and compiler are placed inside the HAL subfolder. Each of these file names starts with the "__HAL_" prefix, followed by the indexed compiler or architecture name, or both of them combined.

Demo applications, composed of the source files - in addition to the project files, are placed inside the "example" folder, organized by subfolders, named with the same suffixes as used for the HAL. Only ARM folder has additional subfolders.



Packages for the MikroElektronika compilers are placed inside the package folder, organized by subfolders, named by the programming language. Package files suffix consists of indexed compiler name - similar to HAL, but in this case, there is only one package for the ARM compilers, which contains packages for all the supported vendors.

Documentation is placed inside the doc folder and besides the doxyfile (useful for generation of the HTML help documentation), there is Windows OS help format file, which is built using the additional .css file - with customized header and footer. Windows help file is always named "helpfile.chm"


packages
C
ARM
PIC
basic
ARM
PIC
doc
doxy
Doxyfiledoxy
CSS
tex
img
...
schematic.pdf

Source coding rules

Rules are divided by meaning and not all of these rules must be strictly followed. The most important are the implementation rules, while syntax and naming rules are not crucial - but there are few of them which must be followed, in order to have the library usable on more compilers.

Key rules

1. All library global accessible identifiers must start with the unique prefix
2. Library must be written in pure ANSI C89, without the usage of any compiler specific function
3. Delay functions are the only functions which are allowed, but only the predefined ones - with no argument
4. All publicly visible types must be a type of <stdint.h> library or at least typedef derived by some of stdint.h types
5. Use unsigned types whenever possible

Other rules

Functions

1. Global function identifier must start with a unique library prefix
2. Private function identifier should not have library prefix
3. Function identifier should contain at least one additional word separated with the character " _" - verb which comes right after the prefix
4. Exception to this rule are initialization functions, which are special kind of functions
5. Usage of "_" character inside argument identifier is forbidden
6. Argument should be a noun - one word if possible
7. Counter argument identifier should start with the n prefix
8. Pointer argument identifier should start with the p prefix
9. Function pointer argument identifier should start with the fp prefix
10. Braces come right after the function identifier, without any spaces in between
11. Use spaces between argument indenters but also between identifiers and braces

```
uint8_t click_gpioDriverInit( T_CLICK_P gpioObj );
uint8_t click_setHandler( T_click_handler *fpHdl );

uint8_t function( uint8_t key, uint8_t key2 )
{
    uint8_t result;
    result = key + key2;
    return result;
}
```

```
void click_readRegister( uint8_t *pData, uint8_t nBytes )
{
    while (nBytes--)
    {
        readData( *(pData++ ) );
    }
}
```

Explanation: function naming rule allows having easily recognizable function names. Function argument naming rule, in addition to variable naming rule, allows for having a prediction of what exactly the argument is - and to avoid naming collision between arguments and variables.

Variables

12. Prefix is mandatory inside variable identifier for all global accessible variables and defines
13. All global accessible variable identifiers have "_" character as prefix
14. Constant variable is written using only capital letters
15. Variable identifier should be a noun, which can be extended using an adjective as prefix
16. Extern variables must have library prefix, but usage of "_" character prefix is forbidden
17. Each variable which is candidate for function argument must be a constant
18. Variables such counters, flags and pointers, should have appropriate prefix, same as in the case of function arguments, but separated with the "_" character, from the noun.
19. Every declared static variable should be set to initial value during module initialization call
20. Every local variable must be declared at the beginning of the block

```
const uint16_t _CLICK_DEFAULT_STATUS = 0x15;

uint8_t slaveAddress;
bool f_busy;
T_click_hanlder *fp_defaultHandler;

void function( uint8_t input )
{
    uint16_t n_tick = 0;

    slaveAddress = 0x15;
    f_busy = false;

    while (n_tick++ < input)
    {
        if (gpio_pin())
        {
            f_busy = true;
        }
    }
}
```

Explanation: ANSI C89 standard requires declaration of variables at the begging of the block, so if the code needs to be compatible with such compilers, those rules have to be followed.

Macros & defines

21. Each macro or define is named using only capital letters
22. Macros and defines placed inside the source file, should not have library prefix
23. Defines necessary for the higher layer, must be named with the starting library prefix
24. Usage of defines as arguments for functions is not allowed

Explanation: macros and defines are pre-processors, so they cannot be provided through the compiler object files. mikroPascal and mikroBasic compilers use object files generated by the mikroC compiler, so the macros and defines won't be accessible by those compilers.

Structures and enums

25. Usage of global accessible types, should be avoided whenever possible
26. Exception to previous is type derived from stdint.h library types
27. Usage of "T_" prefix is mandatory for structure and enum type identifiers
28. Each global accessible type must have library prefix after "T_" prefix and before additional content
29. Structure and enum field identifier should be a noun in addition of an adjective, whenever needed
30. Structure member naming should be same as the variable naming rule.
31. Enum members naming should follow macros and defines rules.

```
typedef struct
{
    uint8_t sid;
    uint8_t eid[2];
    uint8_t remoteRequest;
    uint8_t senderAddress;
    uint8_t payload[64];

}T_MCP25625_message;

typedef enum
{
    MCP25625_POS_LEFT,
    MCP25625_POS_RIGHT,
    MCP25625_POS_UP,
    MCP25625_POS_DOWN

}T_MCP25625_position;
```

Explanation: all defined types cannot be provided through the .mcl or .emcl compiler object files, so whenever enum is needed, it is better to define constant, which will be named according to constant variable naming rule, rather than a member of enum.

Conditional statements

32. Conditional statement key word should be separated with space from

braces related to condition

33. Continue statements are forbidden
34. Break statement should be avoided whenever possible
35. Instead of switch statement, if and else should be used
36. Goto is forbidden
37. Same as for functions, block braces comes to the next line
38. Single line block without braces is forbidden

```
uint8_t function( uint8_t key, uint8_t key2 )
{
    uint8_t result;
    if (key > key2)
    {
        result = key;
    }
    else
    {
        result = key2;
    }
    return result;
}
```

Explanation: ANSI C89 allows usage of constants as case labels, while according to C99 that's not allowed. Goto, break and continue statements may lead to code structure that is hard to read.

Documentation rules

1. Documentation is written inside the header file
2. Each public member must be documented, according to the Doxygen documentation rules
3. All function arguments should be documented, in addition of the direction tag
4. Group of variables or functions should be placed inside the group tag
5. Documented structures, enums, typedefs and macros, should have appropriate tag in addition of brief description

```
/** @file my_file.h */

/** @defgroup CL_REGS Registers */
/** @{ */

const uint16_t CLICK_REG_1 = 0; /**< Register 1 */
const uint16_t CLICK_REG_2 = 1; /**< Register 2 */

/** @} */

/** @struct T_click_colors
 *  @brief Colors
 */
```

```
typedef enum
{
    CLICK_COLOR_RED,
    CLICK_COLOR_BLACK
    CLICK_COLOR_WHITE

}T_click_colors;

/**
 *  @brief Function Example
 *
 *  @param[in] inPar input parameter
 *  @param[out] outPar output parameter
 *  @retval 0 successful operation
 *
 *  Additiona desription...
 */
uint8_t function( uint8_t inPar, uint8_t *outPar );
```

SUMMARY

The mikroSDK standard defines a set of coding rules. Following these rules ensures that the library can retain its portability among other platforms and architectures.

Practical Usage

of the mikroSDK in MikroE compilers

In the following section, the practical usage of the mikroSDK will be demonstrated. There are several simple steps that needs to be taken, before starting to work with the mikroSDK:

- The mikroBUS API **.mpkg** package for the used compiler and the **.mpkg** library for the desired click board need to be downloaded and installed.
- After the installation, both of them should appear as new nodes, inside the Library Manager of the used compiler.
- Right click to the newly installed click library item in the Library Manager and click Examples to open examples folder.
- Double click to project file to open it.

This will load project for the click board inside the working environment. The examples provided are already configured for the default development systems:

– PIC Compilers

+ EasyPIC PRO v7 + P18F87K22

– PIC32 Compilers

+ EasyPIC Fusion v7 + P32MX795F512L

– dsPIC Compilers

+ EasyPIC Fusion v7 + P33FJ256GP710A

– AVR Compilers

+ EasyAVR v7 + ATMEGA32

- FT90x Compilers

+ EasyFT90x + FT900

– ARM Compilers

+ EasyMX PRO v7 for SMT32 + STM32F107VC (STM32 Vendor)
+ EasyMX PRO v7 for TIVA + TM4C129XNCZAD (TI Vendor)
+ HEXIWEAR docking station (KINETIS Vendor)
+ Clicker 2 for CEC1702 (Microchip Vendor)
+ Clicker 2 for MSP432 (TI Vendor)

So, if any of these systems are used, all it needs to be done is:

Go to Build > Rebuild all sources (Alt + F9)

Go to Build > Build + Program (Ctrl + F11)

This will program the used MCU and the application demo example will work in out of box manner, without any additional interventions on the code. The click board should be inside the proper mikroBUS™ socket which can be clearly seen in the system and application Init functions, in most cases it is the first socket.

In cases that some other development system supported by the mikroSDK is used:

Select the proper MCU inside “Project settings” and the used system will appear in Library Manager - check it

Go to Project > Edit project and load proper scheme for the used MCU (Ctrl + Shift + E)

Go to Build > Rebuild all sources (Alt + F9)

Go to Build > Build + Program (Ctrl + F11)

If there is a need to change the used mikroBUS™ socket, all it needs to be done, is to change few characters inside system Init and application Init functions inside the application demo example.

DOCUMENT HISTORY

December 20, 2017 - version 1.0 – Initial release

SOFTWARE LICENSE

mikroSDK

mikroe.com/mikrosdk

mikroe.com/mikrobus

libstock.mikroe.com