AFBR-S50 SDK - Argus API Reference Manual - v1.2.3

AFBR-S50 Time-of-Flight Sensor SDK for Embedded Software

Chapter 1

Introduction

The *AFBR-S50 SDK* is the appertaining software for the AFBR-S50 Time-of-Flight Sensor family by Broadcom Inc. The SDK consists of evaluation tools running with the Evaluation Kit as well as the *AFBR-S50 Core Library* and its API for the sensor devices.

The evaluation tools contain an fully implemented reference solution for the *AFBR-S50 Core Library*. It utilizes the corresponding software API and provides another, equivalent hardware API in order to access and control the sensor device through a serial hardware interface, such as USB or UART. The *ExplorerApp* is running on a Cortex-M0+ processor such as the NXP Kinetis L-Series (e.g. MKL46z is used for the Evaluation Kit). In order to connect and evaluate the sensor, a PC GUI, the *AFBR-S50 Explorer*, is provided. The GUI establishes a serial connection via USB to the *ExplorerApp* running on the *MKL46z Cortex-M0+ by NXP* and leverages the API and core library to apply configuration and obtain measurement results. The latter is conveniently visualized in 1D or 3D plots.

1.1 API Overview

The *AFBR-S50 SDK* contains the sensor API that wraps around the core library and provides an easy access to the sensors vast functionality to the user. A very simple example demonstrates the basic usage of the API and serves as a starting point for any user implementation. A comprehensive reference implementation (namely the *ExplorerApp*, utilized for the evaluation kit) unveils the complete capability of the API and coats it in a serial hardware interface that connects via USB to the *AFBR-S50 Explorer* GUI. The latter can easily adopted to other serial interfaces such as UART.

The *AFBR-S50 Core Library* is required to operate the *AFBR-S50 Sensor Family* along with the users application on the same target platform. It is a comprehensive, yet simple library that implies all the algorithms that are required to run the sensor with nearly any microprocessor that is based on the ARM Cortex-Mx Architecture. Therefore the library is implemented hardware independent and the user needs to establish the connection from the library to the underlying hardware by implementing the provided hardware interface layers for its platform. Apart from that, the library is easy to use and runs basically in the background through interrupt service routines without putting much load on the main thread such that the CPU remains available and responsive for the user application.

There are example implementations of the *AFBR-S50 Software API* that serve, along with the provided platform drivers, as a starting point for user implementations. All the examples are build on the *Cortex-M0+ Architecture* in general and on the NXP Kinetis MKL46z (Evaluation Kit) and NXP Kinetis MKL17z (Reference Design).

1.2 Getting Started

If you want to implement and utilize the *AFBR-S50 Core Library* and API within your own embedded environment, go to the the getting started section. Make sure to read about the examples and how to get them running with the free `MCUXpresso IDE from NXP` in the how to build and run section. Also see the Software API section for an brief overview of the API functionality.

1.3 Porting to a new MCU platform

If you want to use your own hardware to host and run the *AFBR-S50 Core Library*, refer to the API Porting Guide.

1.4 Copyright

Chapter 2

Architectural Overview

2.1 Overview

See the Fig. 2.1 for an brief overview on the integration of the *AFBR-S50 Core Library* and API into the users application and hardware platform. The architecture is split into two parts: hardware and software. The hardware contains the *AFBR-S50* sensor device. It is connected to the processor and its peripherals via an SPI interface. An additional GPIO pin is required for the measurement data ready event from the sensor device.

The software that runs on the processor is essentially build from three parts: The *user application*, holding application specific code, is implemented by the user. The *AFBR-S50 Core Library* consists of sensor hardware specific code, an corresponding API and the hardware abstraction layers (HAL). The latter defines the required interface for the *AFBR-S50 Core Library* to access the hardware and is implemented platform dependent in the user application.

The *AFBR-S50 Core Library* comes as a pre-compiled *ANSI-C* library (i.e. a "lib$*$.a" file) file that contains all data and algorithms required to run the sensor hardware. It is wrapped within an API that allows the user to access the internal functionality of the core library to control the device. The user application will basically access the core functionality via function calls through the API and has further the opportunity to install callbacks within the core in order to get informed about events in the library, e.g. new measurement data ready or other status changes.

Since the *AFBR-S50 Core Library* requires access to the hardware peripherals to communicate with the sensor device. Dedicated platform interfaces are defined by the API to be implemented by the user based on the underlying

hardware platform.

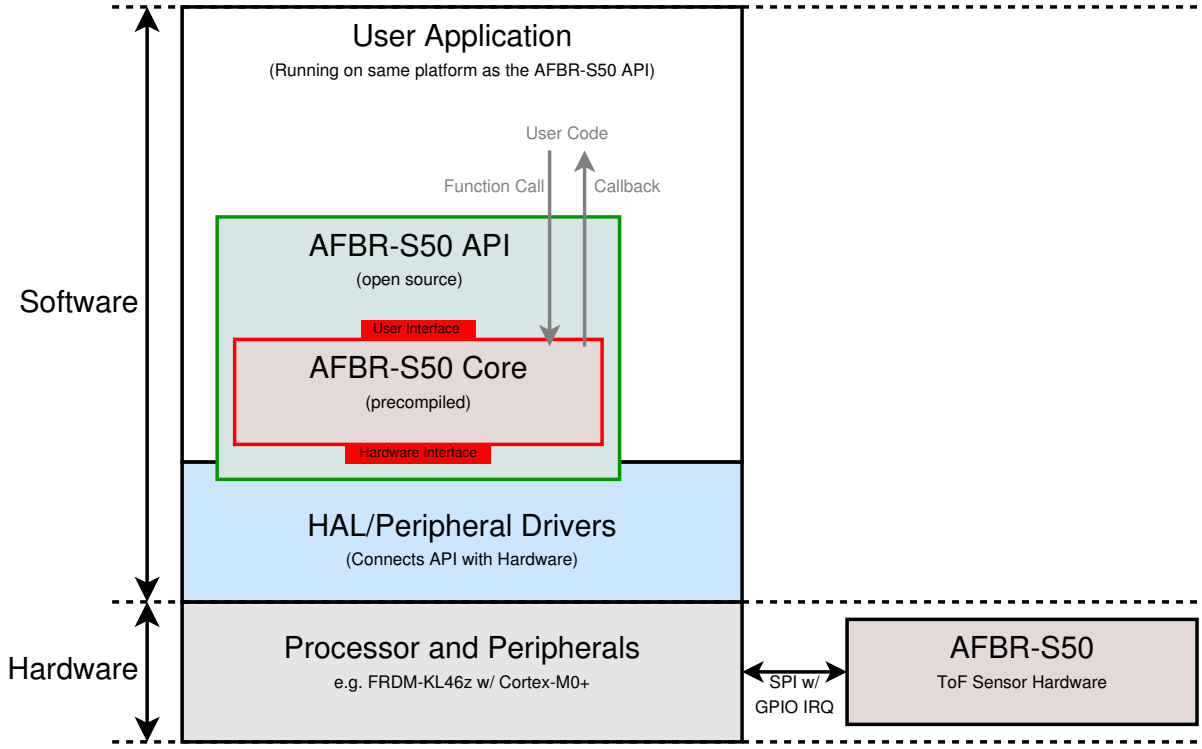


Fig. 2.1: An overview of the ∗AFBR-S50 SDK∗ architecture for integration into an user application.

2.2 Operation Principle

The basic operation principle of the *AFBR-S50 Core Library* is described below:

- Measurement routines are running asynchronously in the background and notify the user application via a callback when a measurement has finished and new raw measurement data is ready to be used. The measurements are triggered either by a periodic timer interrupt at a fixed frame rate or by a function call from the user application. The actual measurement sequence is driven by short interrupt service routines and therefore the main processor tasks are not blocked while measurements are ongoing.
- The evaluation of the raw measurement data is executed independently of the measurement activity. When the main task is informed about the new raw measurement data ready event from the core by callback, it can call the evaluation routine whenever appropriate. However, the raw data buffer is blocked until the data is evaluated. Thus, not calling the evaluation routine blocks the start of new measurement frames. A double buffer architecture is utilized in order to allow interleaved execution of measurements and evaluation tasks.
- After calling the evaluation routine, the measurement data is ready and calibrated. The raw data buffer is empty and ready to be used with a new measurement frame to be performed on the device.
- In order to adopt to changing operating conditions, new configuration or calibration parameters can be applied at any time; the API manages to write updated configuration data between two measurement frames.
- The Dynamic Configuration Adaption (DCA) feature is provided to automatically adopt to changing environment parameters, such as target reflectivity, ambient light or temperature. This feature highly increase the dynamic measurement range.

- The final evaluated measurement data obtained from the API consists of 3D data, i.e. range and amplitude per pixel, as well as 1D data, i.e. range and amplitude evaluated from a certain number of pixels. The selection of the pixels to determine the 1D data is realized via the Pixel Binning Algorithm (PBA) module.
- Calibration algorithms are applied within the evaluation sequence and to the device configuration in order to compensate environmental influences or device-to-device deviations. Each device comes with its individual factory calibrated values in an EEPROM that is read and applied in by the Calibration module.
- *Auxiliary* measurement tasks are utilized to obtain information about the system health and the environment and adopt the system correspondingly. These measurements are automatically executed in the background after each measurement cycle.
- A *Dual-Frequency Mode* is provided to overcome the limited unambiguous range issue and remove ghost pictures.
- *Eye-Safety* is derived from the static default configuration and adopted to any configuration changes made via the API. Also the DCA module heeds the laser safety limits. A timeout will watch the system responsiveness and the *Reference Pixel* is used to monitor the health state of the laser source. A system shut down is triggered in case of any failure, e.g. a laser short circuit.

2.3 API Modules

The *AFBR-S50 Core Library* contains a comprehensive set of routines and algorithms to run the *AFBR-S50 Time-of-Flight Sensor* devices. The functions and objects are defined and implemented in the AFBR-S50 Main API module. It consists of several submodules that handle specified tasks, such as

- device configuration,
- device calibration,
- measurement data evaluation or
- generic/measurement operation.

In order to provide a portable API library, the platform specific driver and HAL implementations are not included into the library. Instead, the connection to the hardware is obtained via the implementation of provided interfaces for the required hardware and peripherals. In detail, the hardware requirements are:

- S2PI: The communication with the device requires an SPI interface. In order to increase speed and lower the CPU load, it is recommended to us a DMA module along with the SPI interface. In order to get informed about the measurement data ready event, a single GPIO IRQ is required and incorporated into the SPI module. In order to decrease the complexity for connections, the devices EEPROM is connected to the SPI pins but speaks a different protocol. In order to enable the calibration data readout from the EEPROM, the SPI pins must be accessed as GPIO pins in order to emulate the protocol by software `bit banging` algorithms. All these, i.e. SPI and GPIO, are summarized in the S2PI module.
- Timer: In order to heed the laser safety limits, a timer for time measurements concerns is mandatory. Note that this timer must be setup carefully in order to guarantee the laser safety to be within Class 1. Furthermore, a periodic interrupt timer can be used to trigger measurements autonomously in the background on a time based schedule. This is optional, since for simple implementations it might be sufficient to start the measurements on demand from the foreground threads by calling the corresponding API function.
- IRQ: In order to prevent race conditions and such, the program must be able to lock the occurrence of interrupts in atomic or critical sections of the code.
- Logger: Optionally, the logger module can be implemented and utilized to obtain debug information from the API operation. This is done by a *printf*-like function.

- Non-Volatile Memory: If user calibration is used, it is recommended to implement the interface to access a non-volatile memory peripheral (e.g. flash memory). If implemented, the user calibration data (e.g. crosstalk vectors) can be stored within the *Core Library* and will be automatically reloaded after device initialization (e.g. after a power cycle). If not implemented, the user can manage the calibration and configuration via the API.

Finally, there are some utilities implemented to help the user to adopt to the API architecture:

- Fixed Point Math: A small and effective fixed-point library containing essential algorithms and definitions.
- Time Utilities: Methods to handle the specified time format provided by the API.
- Misc. Utilities: Miscellaneous Integer Math like long (64-bit) multiplication or integer square root.

Chapter 3

Getting Started

The following section gives an brief overview on of the **AFBR-S50 Core Library and API** and shows how to get started using the evaluation platform and example files. If a port to another microcontroller platform is required, refer to the Porting Guide.

3.1 AFBR-S50 API

Note

All **AFBR-S50 API** related functions, definitions and constants have a prefix "∗∗Argus∗∗" which is essentially an alias or working title for the **AFBR-S50 Time-of-Flight Sensor** device.

The *AFBR-S50 Core Library* is provided as a static ANSI-C library file ("lib∗.a") and the corresponding API is provided as ANSI-C header files ("∗.h"). After setting up the linker to link the library, it is sufficient to include the main header in the "/include" folder, "argus.h":

```
#include "argus.h"
```

The API utilizes an abstract handler object that contains all internal states for a single time-of-flight sensor device. In this way, it is possible to use the same API with more than a single device. Note, however, that this feature is not fully implemented in the current version and is planned for a future release! After including the header file, the handler object must be created by calling the Argus_CreateHandle function to obtain a pointer to the newly allocated object. This is done via the standard library function:

```
void * malloc(size_t size)
```

If it is required to use a different function, create and overwrite the weakly linked method and implement your own memory allocation algorithm:

```
void * Argus_Malloc(size_t size) { /* ... */ }
```

After creation of the handler object, the *AFBR-S50* module must be initialized with the corresponding handler object:

```
status_t status = Argus_Init(hnd, SPI_SLAVE);
```

Note that all peripheral modules must be ready to be used before executing any API function. So make sure to initialize the board and its peripherals before. Now the device is ready to run and has setup with default configuration and calibration data. Use the provided API functions to customize the given default configuration to the needs and requirements of the application.

From then, there are two possibilities to operate the device. First, the simple polling method. The measurements are triggered by the main program via calling the Argus_TriggerMeasurement function any time a new measurement should be started. A new measurement frame is started and after reading the data from the device, the callback is invoked to inform the host application about the data ready event. In the meantime, the host application can

poll the module status or execute other tasks. After finishing the measurement, the Argus_EvaluateData function must be called to obtain measurement data like range from the raw readout data. It is mandatory to call the evaluation method. Otherwise, the raw data buffer is kept occupied and no new measurement can be started anymore. The module contains however a double buffer architecture, which allows to start the next measurement and evaluate the current measurement data while the device executes a new measurement frame. After evaluation, the argus_results_t data structure is filled with all measurement results that can be processed now be processed by the host application depending on the users needs. An example implementation is shown in the Simple Example section.

Please note that the laser safety module might refuse to restart a measurement at the time the function is called. This is due to timing constraints dictated by the laser safety rules. The frame time and similar parameters can be adjusted with the configuration API methods.

The second way of operating the device is to leverage from an periodic interrupt time that invokes a callback to the API in periodic manner. The timer is implemented in the timer interface. Instead of calling the Argus_TriggerMeasurement function periodically from the host application, the measurement restarts itself in an autonomous way. Every time, a new raw measurement data set is ready, the measurement data ready callback is invoked by the API. Similar to the previous method, the Argus_EvaluateData function must be called before the data can be used. Note that not calling the function will lead to measurements are not restarted before the evaluation method is called and the data buffers is freed. In the same manner, a slow data evaluation or much user code to delay the data evaluation method might decrease the measurement frame rate. An example implementation is shown in the Advanced Example section.

3.2 Simple Example

Here is an example of how to use the API in a simple loop with polling the module status to wait for the measurement data to be ready. The 1D range value of the obtained measurement data is streamed via an UART connection. Open a terminal (e.g. `Termite`) and open a UART connection using 115200 bps, 8N1, no handshake connection. Range values will start to occur on the terminal as soon as the program starts its execution.

```
/*****************************************************************************
 * Include Files
 *****************************************************************************/
#include "argus.h"
#include "board/clock_config.h"
#include "driver/cop.h"
#include "driver/gpio.h"
#include "driver/s2pi.h"
#include "driver/uart.h"
#include "driver/timer.h"
/*****************************************************************************
 * Defines
 *****************************************************************************/
#define SPI_SLAVE 1
#define SPI_BAUD_RATE 6000000
/*****************************************************************************
 * Variables
 *****************************************************************************/
static volatile void * myData = 0;
/*****************************************************************************
 * Prototypes
 *****************************************************************************/
/*!*************************************************************************
 * @brief   printf-like function to send print messages via UART.
 *
 * @details Defined in "driver/uart.c" source file.
 *
 *          Open an UART connection with 115200 bps, 8N1, no handshake to
 *          receive the data on a computer.
 *
 * @param   fmt_s : The usual printf parameters.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ***************************************************************************/
extern status_t print(const char  *fmt_s, ...);
/*!*************************************************************************
 * @brief   Initialization routine for board hardware and peripherals.
 *
 * @return  -
 ***************************************************************************/
```

```
static void hardware_init(void);
/*!*****************************************************************************
 * @brief   Measurement data ready callback function.
 *
 * @details
 *
 * @param   status :
 *
 * @param   data :
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t measurement_ready_callback(status_t status, void * data);
/*****************************************************************************
 * Code
 *****************************************************************************/
/*!*****************************************************************************
 * @brief   Application entry point.
 *
 * @details The main function of the program, called after startup code
 *          This function should never be exited.
 *
 * @return  -
 ******************************************************************************/
int main(void)
{
    /* The API module handle that contains all data definitions that is
     * required within the API module for the corresponding hardware device.
     * Every call to an API function requires the passing of a pointer to this
     * data structure. */
    argus_hnd_t * hnd = Argus_CreateHandle();
    if (hnd == 0)
    {
        /* Error Handling ...*/
    }
    /* Initialize the platform hardware including the required peripherals
     * for the API. */
    hardware_init();
    /* Initialize the API with default values.
     * This implicitly calls the initialization functions
     * of the underlying API modules.
     *
     * The second parameter is stored and passed to all function calls
     * to the S2PI module. This piece of information can be utilized in
     * order to determine the addressed SPI slave and enabled the usage
     * of multiple devices on a single SPI peripheral. */
    status_t status = Argus_Init(hnd, SPI_SLAVE);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    /* Adjust some configuration parameters by invoking the dedicated API methods. */
    Argus_SetConfigurationFrameTime(hnd, 100000); // 0.1 second = 10 Hz
    /* The program loop ... */
    for(;;)
    {
        myData = 0;
        /* Triggers a single measurement.
         * Note that due to the laser safety algorithms, the method might refuse
         * to restart a measurement when the appropriate time has not been elapsed
         * right now. The function returns with status #STATUS_ARGUS_POWERLIMIT and
         * the function must be called again later. Use the frame time configuration
         * in order to adjust the timing between two measurement frames. */
        status = Argus_TriggerMeasurement(hnd, measurement_ready_callback);
        if (status == STATUS_ARGUS_POWERLIMIT)
        {
            /* Not ready (due to laser safety) to restart the measurement yet.
             * Come back later. */
            __asm("nop");
        }
        else if (status != STATUS_OK)
        {
            /* Error Handling ...*/
        }
        else
        {
            /* Wait until measurement data is ready. */
            do
            {
                status = Argus_GetStatus(hnd);
                __asm("nop");
            }
            while(status == STATUS_BUSY);
            if (status != STATUS_OK)
            {
                /* Error Handling ...*/
            }
```

```
            else
            {
                /* The measurement data structure. */
                argus_results_t res;
                /* Evaluate the raw measurement results. */
                status = Argus_EvaluateData(hnd, &res, (void*)myData);
                if (status != STATUS_OK)
                {
                    /* Error Handling ...*/
                }
                else
                {
                    /* Use the recent measurement results
                     * (converting the Q9.22 value to float and print or display it). */
                    print("Range: %d mm\n", res.Bin.Range / (Q9_22_ONE / 1000));
                }
            }
        }
    }
}
static void hardware_init(void)
{
    /* Initialize the board with clocks. */
    BOARD_ClockInit();
    /* Disable the watchdog timer. */
    COP_Disable();
    /* Init GPIO ports. */
    GPIO_Init();
    /* Initialize timer required by the API. */
    Timer_Init();
    /* Initialize UART for print functionality. */
    UART_Init();
    /* Initialize the S2PI hardware required by the API. */
    S2PI_Init(SPI_SLAVE, SPI_BAUD_RATE);
}
status_t measurement_ready_callback(status_t status, void * data)
{
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    else
    {
        /* Inform the main task about new data ready.
         * Note: do not call the evaluate measurement method
         * from within this callback since it is invoked in
         * a interrupt service routine and should return as
         * soon as possible. */
        myData = data;
    }
    return status;
}
```

3.3 Advanced Example

Here is an example of how to use the API with the autonomous measurement triggering. The 1D range value of the obtained measurement data is streamed via an UART connection. Open a terminal (e.g. Termite) and open a UART connection using 115200 bps, 8N1, no handshake connection. Range values will start to occur on the terminal as soon as the program starts its execution.

```
/*****************************************************************************
 * Include Files
 *****************************************************************************/
#include "argus.h"
#include "board/clock_config.h"
#include "driver/cop.h"
#include "driver/gpio.h"
#include "driver/s2pi.h"
#include "driver/uart.h"
#include "driver/timer.h"
/*****************************************************************************
 * Defines
 *****************************************************************************/
#define ADVANCED_DEMO 1
#define SPI_SLAVE 1
#define SPI_BAUD_RATE 6000000
/*****************************************************************************
 * Variables
 *****************************************************************************/
static volatile void * myData = 0;
/*****************************************************************************
```

```
 * Prototypes
 ******************************************************************************/
/*!*****************************************************************************
 * @brief   printf-like function to send print messages via UART.
 *
 * @details Defined in "driver/uart.c" source file.
 *
 *          Open an UART connection with 115200 bps, 8N1, no handshake to
 *          receive the data on a computer.
 *
 * @param   fmt_s : The usual printf parameters.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
extern status_t print(const char  *fmt_s, ...);
/*!*****************************************************************************
 * @brief   Initialization routine for board hardware and peripherals.
 *
 * @return  -
 ******************************************************************************/
static void hardware_init(void);
/*!*****************************************************************************
 * @brief   Measurement data ready callback function.
 *
 * @details
 *
 * @param   status :
 *
 * @param   data :
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t measurement_ready_callback(status_t status, void * data);
/*******************************************************************************
 * Code
 ******************************************************************************/
/*!*****************************************************************************
 * @brief   Application entry point.
 *
 * @details The main function of the program, called after startup code
 *          This function should never be exited.
 *
 * @return  -
 ******************************************************************************/
int main(void)
{
    /* The API module handle that contains all data definitions that is
     * required within the API module for the corresponding hardware device.
     * Every call to an API function requires the passing of a pointer to this
     * data structure. */
    argus_hnd_t * hnd = Argus_CreateHandle();
    if (hnd == 0)
    {
        /* Error Handling ...*/
    }
    /* Initialize the platform hardware including the required peripherals
     * for the API. */
    hardware_init();
    /* Initialize the API with default values.
     * This implicitly calls the initialization functions
     * of the underlying API modules.
     *
     * The second parameter is stored and passed to all function calls
     * to the S2PI module. This piece of information can be utilized in
     * order to determine the addressed SPI slave and enabled the usage
     * of multiple devices on a single SPI peripheral. */
    status_t status = Argus_Init(hnd, SPI_SLAVE);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    /* Adjust some configuration parameters by invoking the dedicated API methods. */
    Argus_SetConfigurationFrameTime(hnd, 100000); // 0.1 second = 10 Hz
    /* Start the measurement timers within the API module.
     * The callback is invoked every time a measurement has been finished.
     * The callback is used to schedule the data evaluation routine to the
     * main thread by the user.
     * Note that the timer based measurement is not implemented for multiple
     * instance yet! */
    status = Argus_StartMeasurementTimer(hnd, measurement_ready_callback);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    for(;;)
    {
        /* Check if new measurement data is ready. */
```

```
        if(myData != 0)
        {
            /* Release for next measurement data. */
            void * data = (void *) myData;
            myData = 0;
            /* The measurement data structure. */
            argus_results_t res;
            /* Evaluate the raw measurement results. */
            status = Argus_EvaluateData(hnd, &res, data);
            if (status != STATUS_OK)
            {
                /* Error Handling ...*/
            }
            else
            {
                /* Use the recent measurement results
                 * (converting the Q9.22 value to float and print or display it). */
                print("Range: %d mm\n", res.Bin.Range / (Q9_22_ONE / 1000));
            }
        }
        else
        {
            /* User code here... */
            __asm("nop");
        }
    }
}
static void hardware_init(void)
{
    /* Initialize the board with clocks. */
    BOARD_ClockInit();
    /* Disable the watchdog timer. */
    COP_Disable();
    /* Init GPIO ports. */
    GPIO_Init();
    /* Initialize timer required by the API. */
    Timer_Init();
    /* Initialize UART for print functionality. */
    UART_Init();
    /* Initialize the S2PI hardware required by the API. */
    S2PI_Init(SPI_SLAVE, SPI_BAUD_RATE);
}
status_t measurement_ready_callback(status_t status, void * data)
{
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    else
    {
        /* Inform the main task about new data ready.
         * Note: do not call the evaluate measurement method
         * from within this callback since it is invoked in
         * a interrupt service routine and should return as
         * soon as possible. */
        myData = data;
    }
    return status;
}
```

3.4 Build And Run the Examples using MCUXpresso

In order to run the provided example projects using the MCUXpressIDE by NXP, execute the following steps. Please also refer to the getting started guide by NXP in case of any trouble: https://www.nxp.↩com/docs/en/user-guide/MCUXSDKGSUG.pdf

- Download and install the MCUXpresso IDE (recommended v11.1):
  - Go to https://www.nxp.com/design/software/development-software/mcuxpresso-softwar:MCUXpresso-IDE
  - Click on *Download* and register or sign in to download the installer.
  - Install the IDE.
  - Go to https://mcuxpresso.nxp.com for more information.

- Download and import the KL46z SDK into the MCUXpresso IDE v11.1

  – Open MCUXpressIDE, accept the workspace settings by clicking on "Launch".

  – Click on "Download and Install SDKs" on the "Welcome Page".

  – Go to the "Processors" tab and type "MKL46" into the filter field, select the "SDK_2.x_MKL46Z256xxx4" SDK and click "Install".

  – Accepot the licenses and click on "Finish".

  – After the installation has finished, close the "Welcome" view.

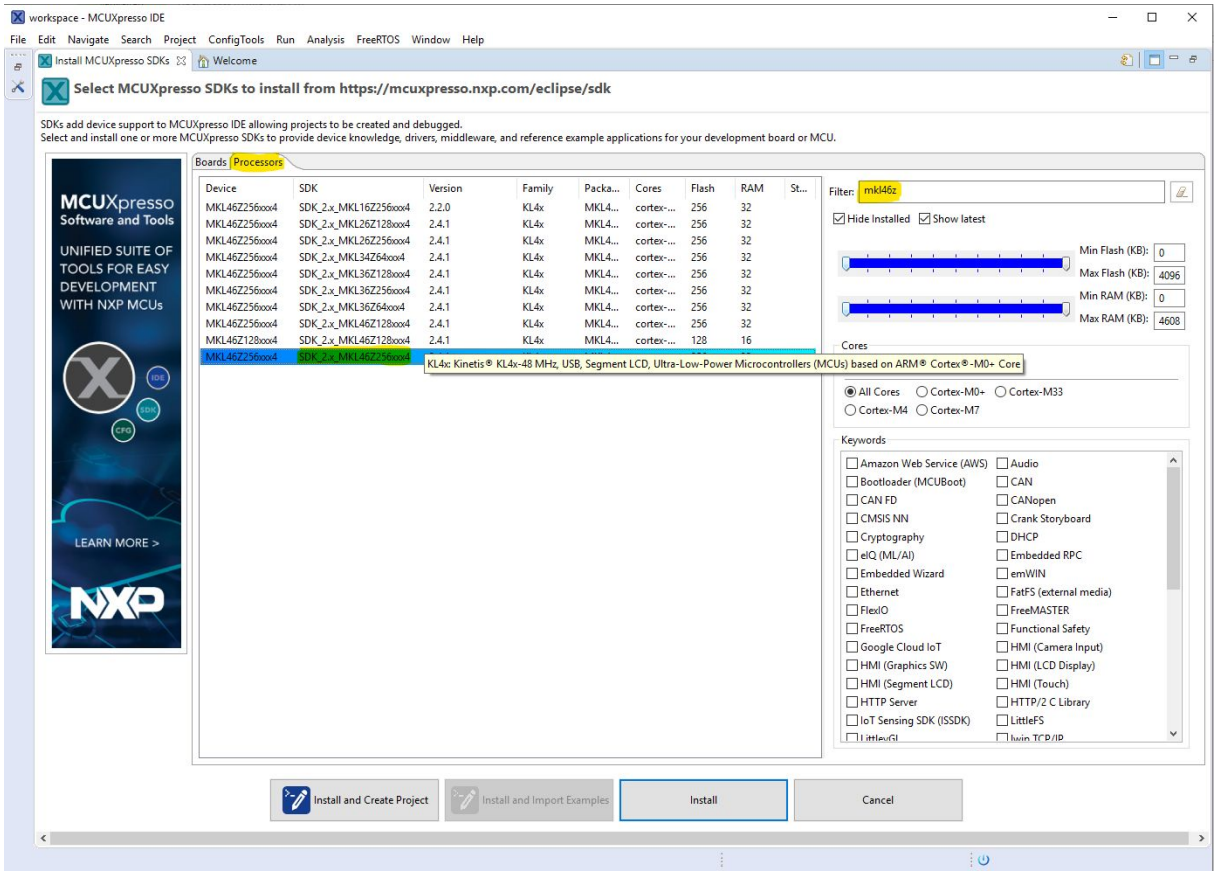


Fig. 3.1: Download and install the SDK files into the MCUXpresso IDE.

- Import the project archive files:

  – Go to MCUXpresso IDE -> Quickstart Panel

  – Click on "Import projects(s) from file system..."

  – Click on "Browse..." in the Archive section

  – Browse to "[INSTALL_DIR]\Device\Projects\" (default is "C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Projects\"

  – Select the required project archive ("∗AFBR_S50_Example_KL46z.zip∗") and click "Open"

  – Click on "Next" -> "Finish"

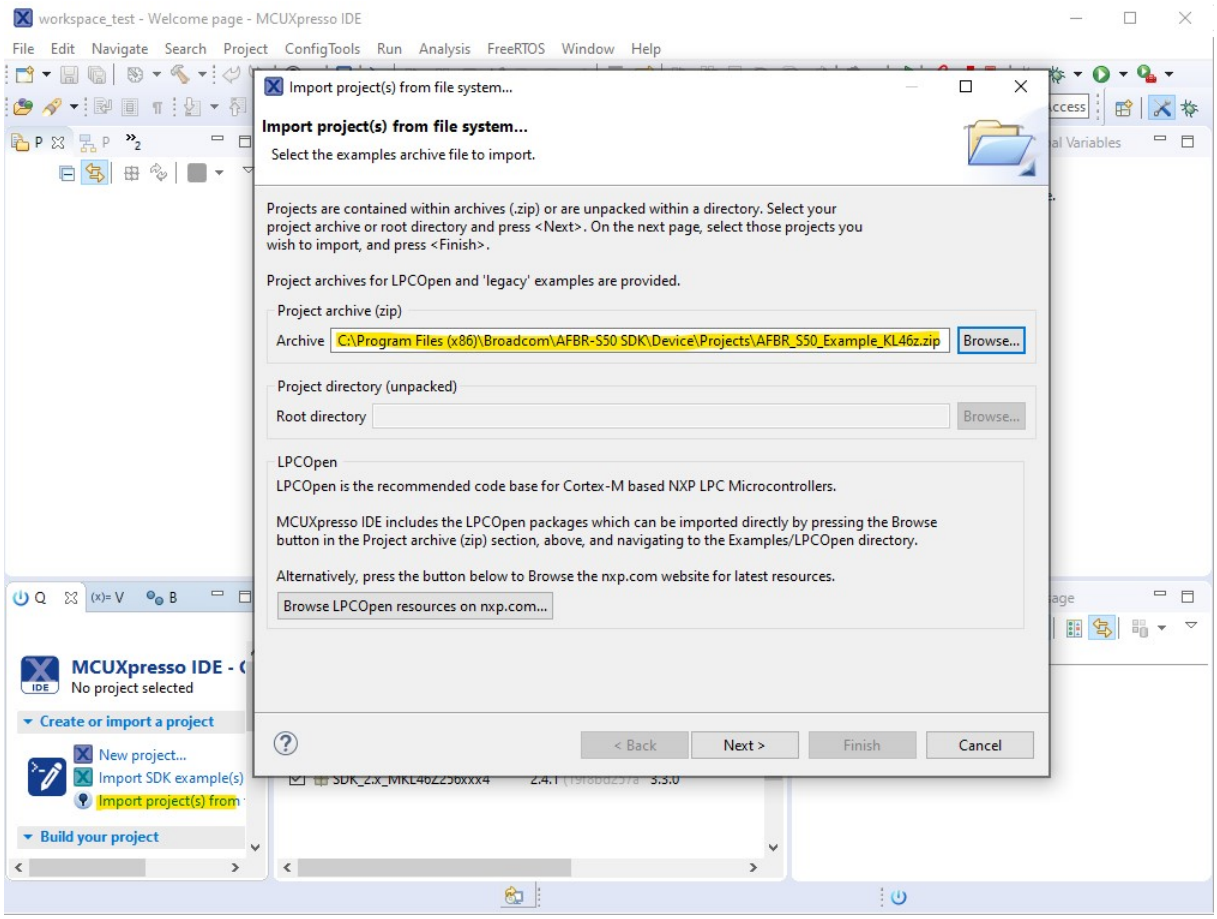


Fig. 3.2: Import the project archive into the MCUXpresso IDE.

- Build the projects:

  - Go to MCUXpresso IDE -> Quickstart Panel

  - Click on "Build"

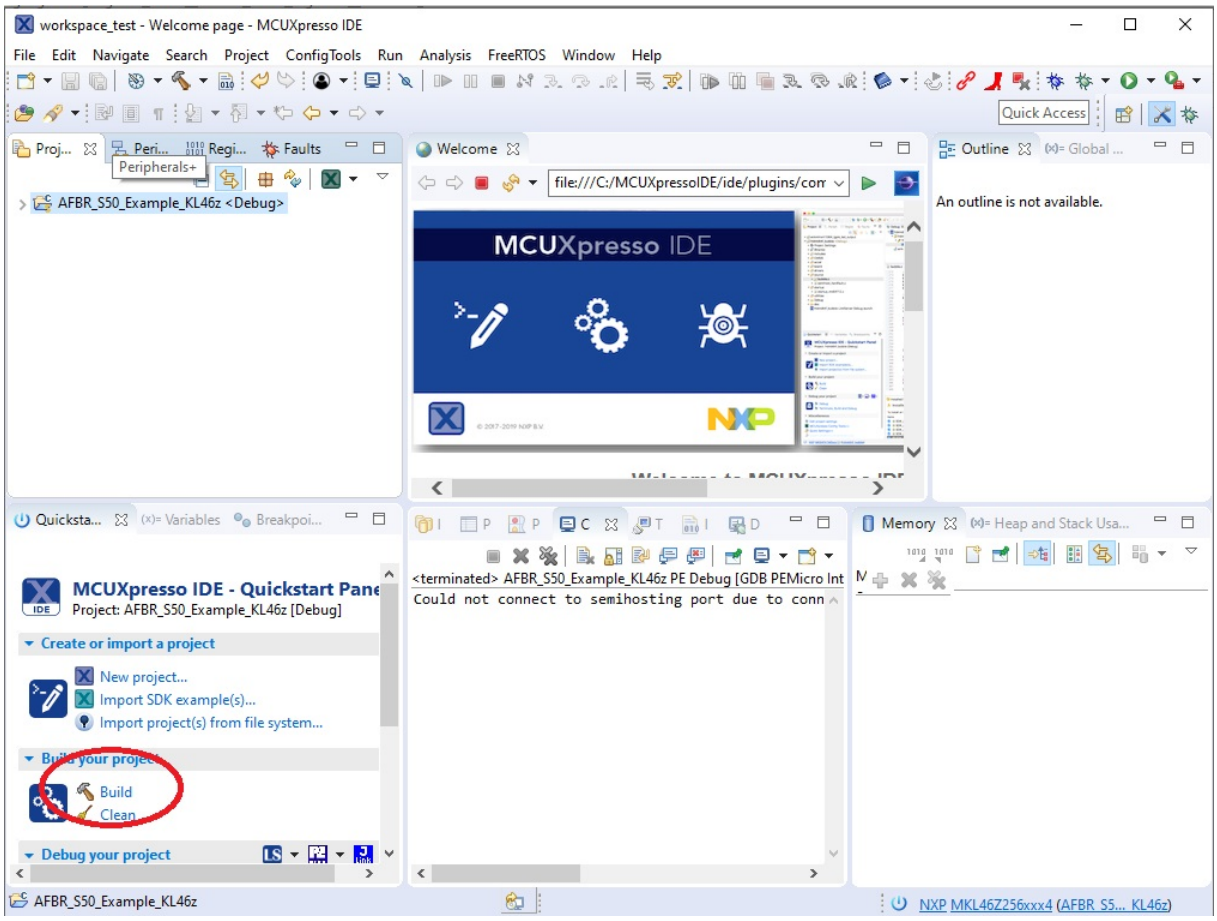


Fig. 3.3: Build the project.

- Download and install the OpenSDA drivers:
  - Go to http://www.pemicro.com/opensda/
  - Download and install *PEDrivers_install.exe*
- Debug and run the project with the OpenSDA debugger:
  - Connect the *OpenSDA* USB port of the KL46z evaluation board.
  - Go to MCUXpresso IDE -> Quickstart Panel
  - Click in the PEMicro Icon (see screenshot).
  - The Debug Probe will be discovered and an according window will show.
  - Click on "OK"

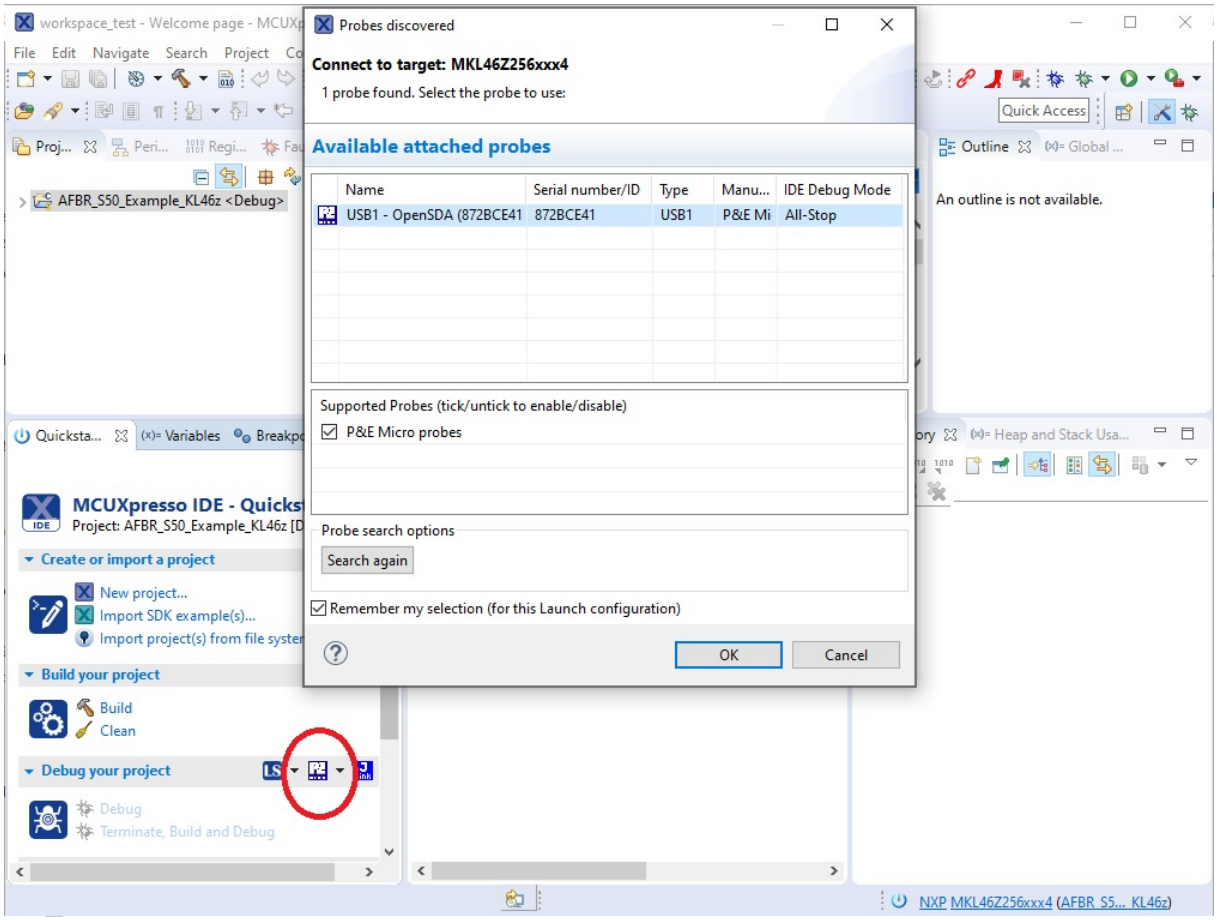


Fig. 3.4: Run and debug the project.

– If the program breaks at the main() function, hit the "Resume" button.

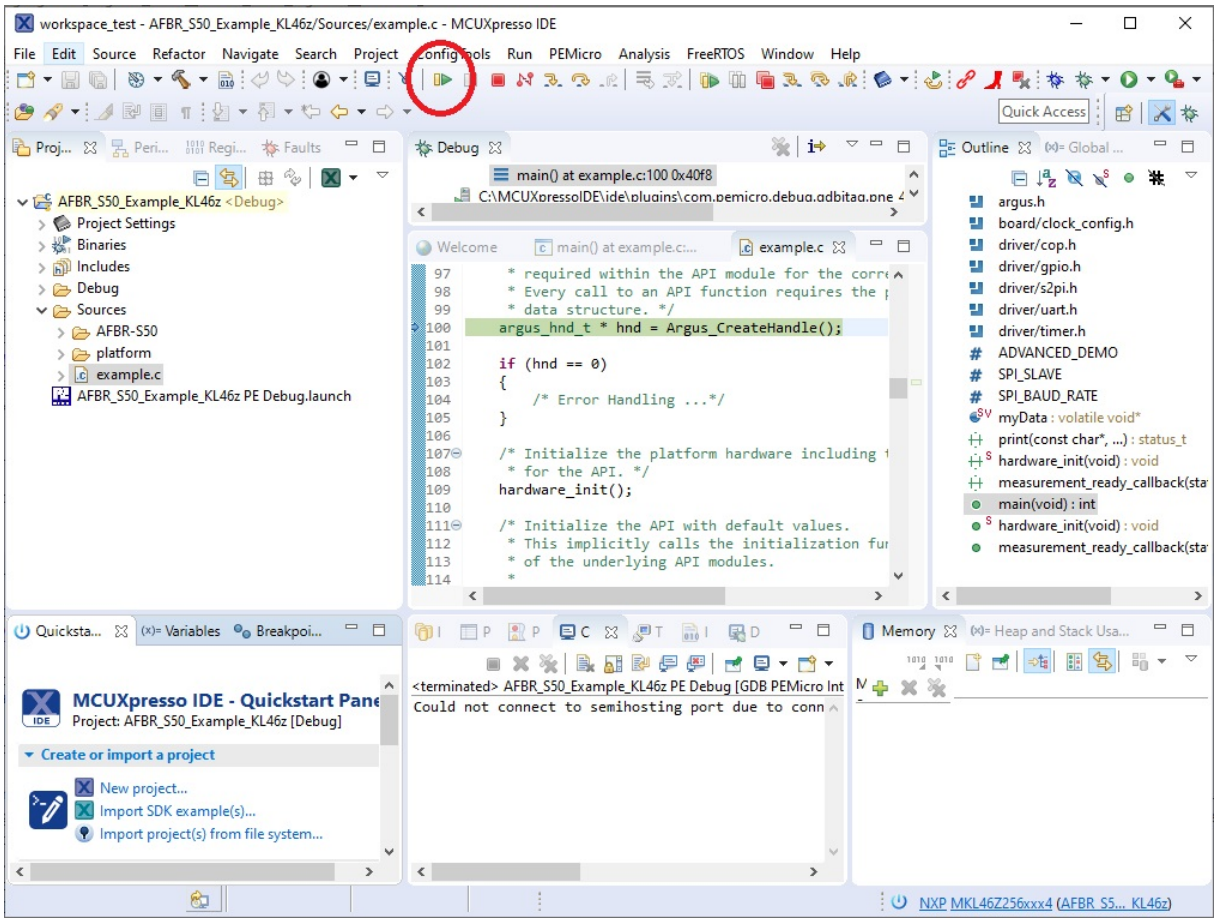


Fig. 3.5: Resume the halted program.

- Display the measurement values on a PC via an UART terminal:
  - Open a terminal (e.g. `Termite`) and open a UART connection using 115200 bps, 8N1, no hand-shake connection.

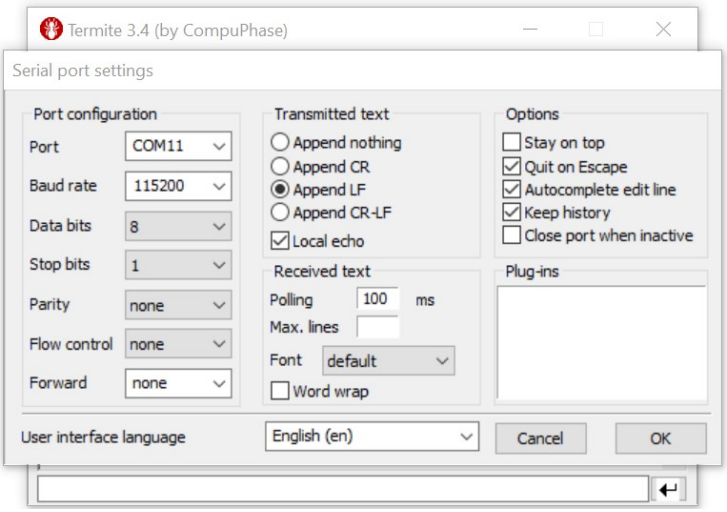


Fig. 3.6: Setting up the terminal to receive measurement results.

  - Range values will start to occur on the terminal as soon as the program starts its execution.

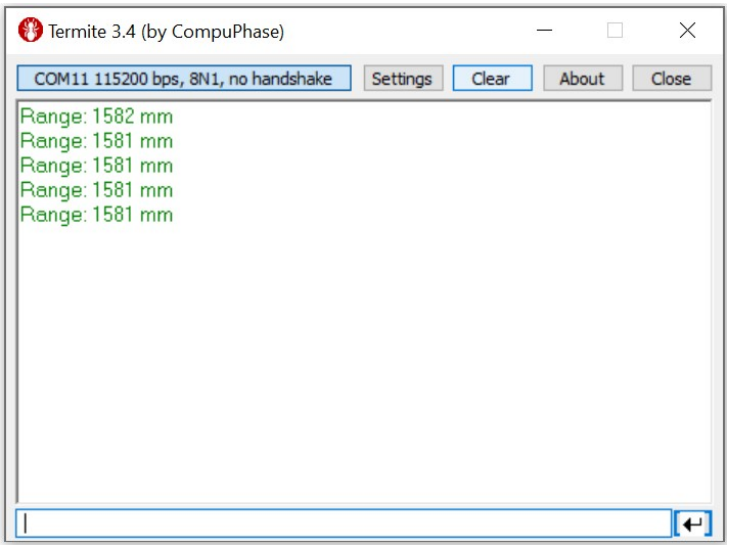


Fig. 3.7: The serial terminal is used to display the received measurement results.

The example application is now up and running. It is a simple program that executes measurements and displays the 1D distance (Units: millimeter) value on the terminal. See also the AFBR-S50 API section for a brief description.

The evaluation kit is build on the FRDM-KL46z Evaluation Kit from NXP. So you may also refer https://www.nxp.com/frdm-kl46z for further information.

3.5 Troubleshooting

3.5.1 Error -101 (Device Not Connected)

Problem: After calling the Argus_Init() method, an error status -101 (ERROR_ARGUS_NOT_CONNECTED) is returned.

The first thing that happens in the initialization function are a few read/write cycles on the SPI interface in order to check the responsiveness of the device: first, a byte sequence (1,2,3,4,...,15) is written to the MOSI and the MISO data is ignored. Second, a sequence of 0's is written to the same register and the MISO is captured and compared to the pattern from the previous write cycle. Finally, another sequence of 0's is written and the received data is checked for being zero again. If the read pattern is not equal to the written one, the error ERROR_ARGUS_NOT_CONNECTED is returned and the initialization process is canceled.

In order to solve the issue, verify the following sequence in your SPI interface when calling the Argus_Init() function:

1. Writing a test pattern to register address 0x04:
   - MOSI: 0x 04 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   - MISO: 0x 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 (or any other, depending on the current state of your device)
2. Clearing the test pattern at register 0x04, read back of test pattern 1:
   - MOSI: 0x 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   - MISO : 0x 00 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

3. Clearing the test pattern at register 0x04, read back of test pattern 2:
    - MOSI: 0x 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    - MISO: 0x 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Further check if the "spi_slave" parameter in the Argus_Init function is not 0! This is reserved for error handling.

Please find the example files in "[INSTALL_DIR]\Device\Examples\" (default is "C:\Program Files (x86)\Broadcom\↩ AFBR-S50 SDK\Device\Examples\").

Chapter 4

MCU Porting Guide

4.1 Introduction

The following section gives an brief overview on how to integrate the *AFBR-S50 Core Library and API* into an user application and how to port the library to another platform. See the Fig. 4.1 for a visualization of the integration progress. The API is embedded into the user application where both are accessing the hardware peripherals via the driver and HAL layers.

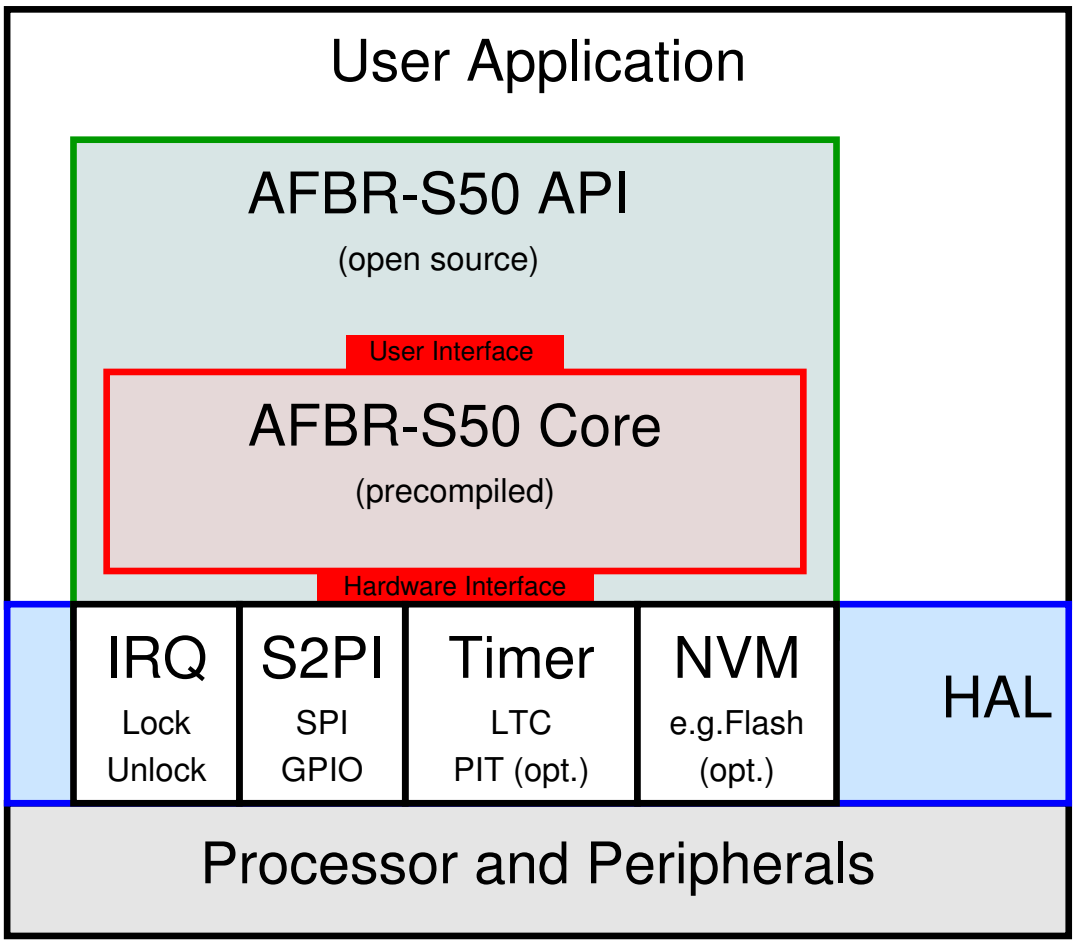


Fig. 4.1: An overview of the AFBR-S50 software architecture for integration into an user application.

The basic idea to port the API to a new platform is to adjust the HAL to the new hardware and implement the functionality that is required by the core library to interface with the *AFBR-S50* sensor device through the given peripherals. The following sections give an overview on the hardware interface layers and finally show a step-by-step guide on how to accomplish the porting task.

4.2 Toolchain Compatibility

The *AFBR-S50 Core Library* is build and tested using the GNU Embedded Toolchain for Arm. However, any other toolchain that supports *AEABI* compatible library linking should work fine with the library files. Note that builds that utilize these toolchains are not tested nor verified and can not be officially supported!

If you still have to use one of these, make sure to read the Binary Interoperability Between Toolchains Application Note by ARM to understand the implications of mixing code from different toolchains.

As mentioned above, the *AFBR-S50 Core Library* is build with *AEABI* compatibility enabled (i.e. with the *-library_interface=aeabi_clib* compiler flag). The following GNU toolchain settings are used to compile the libraries:

- *-library_interface=aeabi_clib*: Specifies that the compiler output is *AEABI* complient.
- *-mfloat-abi=softfp*: Software floating-point Procedure Call Standard (PCS) and hardware floating point instructions.
- *-fno-short-enums*: *enum* types are at least int width.
- *-fno-short-wchar*: the *wchar_t* data type is 4-bytes wide.

Note

The *AFBR-S50* code does not use any floating-point nor *wchar_t* values at all.

4.3 Architectur Compatibility

The *AFBR-S50 Core Library* is compatible with all Arm Cortex-M Series Processors. The library is optimized for the smallest variants, the ∗Cortex-M0/M0+. However, the upwards compatibility of the Cortex-Mx family makes it easy to run the same library on higher architectures without effort as well. The library artifacts are compiled for the most common architectures.

The following API variants are available:

Library Name	Architecture	FPU	ABI	Optimization
afbrs50_m0	Cortex-M0	none	soft	performance
afbrs50_m0_os	Cortex-M0	none	soft	memory size
afbrs50_m0p	Cortex-M0+	none	soft	performance
afbrs50_m0p_os	Cortex-M0+	none	soft	memory size
afbrs50_m1	Cortex-M1	none	soft	performance
afbrs50_m1_os	Cortex-M1	none	soft	memory size
afbrs50_m3	Cortex-M3	none	soft	performance
afbrs50_m3_os	Cortex-M3	none	soft	memory size
afbrs50_m4	Cortex-M4	none	soft	performance
afbrs50_m4_os	Cortex-M4	none	soft	memory size
afbrs50_m4_fpu	Cortex-M4	FPv4-SP	hard	performance
afbrs50_m4_fpu_os	Cortex-M4	FPv4-SP	hard	memory size

Architecture:

- Cortex-M0: -mcpu=cortex-m0
- Cortex-M0+: -mcpu=cortex-m0plus
- Cortex-M1: -mcpu=cortex-m1
- Cortex-M3: -mcpu=cortex-m3
- Cortex-M4: -mcpu=cortex-m4

FPU (Fixed-Point Unit)

- none: No floating-point hardware unit
- FPv4-SP: Hardware floating-point support (-mfpu=fpv4-sp-d16)

ABI (Application Binary Interface):

- soft: Software Floating-point support and ABI (-mfloat-abi=soft)
- hard: Hardware floating-point support and ABI (-mfloat-abi=hard)

Optimization Level:

- performance: The compiler optimized for best performance (-O3)
- memory size: The compiler optimized for lowest memory usage (-Os)

Note

When using another hardware floating point unit, the application must be linked using the *-mfloat-abi=softfp* GNU compiler flag, enabling the software floating-point PCS and hardware floating-point instruction support. The *AFBR-S50 Core Library and API* does not use any floating point operations anyhow.

See also

For more information on the topic see for example this blog.

4.4 Hardware Compatibility

The *AFBR-S50 Core Library* is build and tested using the NXP FRDM-KL46Z development platform that contains a ARM Cortex-M0+ processor. The instruction set available on the *Cortex-M0* is a subset of all the instruction sets available on higher Cortex-Mx processor cores like ARM Cortex-M3 or ARM Cortex-M4. Therefore the library can be used on any *Cortex-Mx* based platforms that contain the required hardware peripherals. These are:

Mandatory hardware peripherals are

- an SPI Interface w/ GPIO access,
- an additional single GPIO IRQ line,

- a lifetime counter that is able to keep track of timing in the magnitude of microseconds during the full lifetime of the embedded system.

Furthermore, optional peripherals are

- a periodic interrupt timer to trigger measurements on a time based schedule by interrupts,
- a non-volatile memory interface (e.g. flash) to save user calibration data beyond a power or reset cycle.

Minimum memory requirements:

- RAM: 8kByte (4kByte Heap + 4kByte Stack)
- ROM/Flash: 128 kByte

Warning

These requirements are the minumum values for the *AFBR-S50 Core Library and API* only! Additional memory for user application is not considered here and must be added accordingly.

The following section gives a brief overview on the corresponding hardware layers.

4.5 Hardware Layers

The AFBR-S50 API basically supports any ARM Cortex-Mx based microcontroller platform. Merely the hardware layers need to be adopted to match the underlying hardware. The API defines interfaces for all required hardware modules and the corresponding methods. Refer Fig. 4.1 to get an overview of the required hardware abstraction layer modules. The interfaces are assembled in the platform module of the API. Each module contains a detailed description on how to implement the corresponding interface. The enclosed example code that comes with the AFBR-S50 SDK also provides implementations of the hardware layers based on the NXP MKL46z platform as a reference. See the getting started section on how to build and run the examples on the provided evaluation platform.

The *AFBR-S50* core has the capability to perform the device operation self-sustained in the background by hopping from interrupt to interrupt without blocking the main processor thread. Therefore all peripheral drivers must be implemented asynchronously, such that an operation is invoked from the API and returns immediately. After finishing the corresponding operation, a callback must be invoked in order to perform the next step in the current sequence. Especially, for the timer interface, there is the requirement of periodic interrupts to be invoked in order to start the measurement cycle in the background without the requirement to call the corresponding function from the application main thread. However, in order to not overload the processor (i.e. from within the interrupt service routines), the data evaluation must be executed from the main thread after the raw measurement data has been read from the device.

4.5.1 S2PI (= SPI + GPIO) Layer

The S2PI module is a combination of SPI and GPIO hardware. The communication with the device requires an SPI peripheral. To increase speed and lower the CPU load, it is recommended to us a DMA module along with the SPI interface. The measurement data ready event occurs when the measurement cycle on the device is finished and the data is ready to read. A single GPIO IRQ invoking a callback to the API core is required and incorporated into the SPI module.

In addition to the standard SPI interface, the corresponding pins must also be accessible in GPIO mode. This is required to access the EEPROM memory of the AFBR-S50 sensor device that holds calibration parameters. The EEPROM interface is connected to the SPI pins to decrease the complexity in pinning and wiring. The EEPROM interface is not compatible with any standard SPI interface and thus it is emulated in software using bit banging algorithms. A mechanism to switch forth and back between SPI and GPIO mode for the corresponding pins is incorporated into the S2PI module.

4.5.1.1 S2PI Overview

The module needs to provide two different modes of operation, both as SPI master:

1. *A fast SPI mode for accessing the device*

   The fast SPI mode is used for accessing the device for all purposes (initialization, configuration, calibration and measurement).

   Obviously, to allow a continuous data transfer on the SPI interface without creating a high load on the microcontroller itself, although not strictly required, it is strongly recommended to set up DMA transfer for the SPI interface. The following description assumes that DMA is used.

2. *A slow SPI mode for accessing the EEPROM*

   Calibration data is stored on a small EEPROM that needs to be read at a much lower speed. The readout is performed on the same interface as the fast SPI, but with a bit-banging mechanism that allows the control of all signals with a much slower timing as GPIOs.

   The bit-banging mechanism is already built into the library, so the hardware layer only has to provide:

   - A mechanism to switch between the two operation modes
   - A mechanism to set the GPIO pins to the required state

   The EEPROM readout is performed only during initialization, so the speed does not negatively affect measurement performance.

4.5.1.2 Initialization

The SPI hardware layer is required to be initialized before the first call to the core library.

4.5.1.2.1 Pin configuration

1. *SCLK, MOSI, MISO*

   The SPI communications requires the three standard pins SCLK (SPI clock), MOSI (master out, slave in) and MISO (master in, slave out) to be switchable between SPI and GPIO modes.

2. *CS*

   Even though it is expected that the device is the only device on the SPI interface, a CS (chip select) signal is required to be asserted on every transfer. Depending on the SPI capabilities, this may be either handled by the SPI implementation (hard CS) or as GPIO (soft CS).

   This signal is active low, and must be reasserted to high between transfers.

3. *IRQ*

   This is an input line that allows the device will signal that measurement data is available and ready for transfer.

   This signal is active low and will be asserted until the next SPI transfer is started, which is assumed to pick up the data. The GPIO should be configured as input triggering an interrupt on the falling edge. Take care that a GPIO with a unique interrupt ID is picked for this line, or no other GPIO attached to the same interrupt ID is configured to trigger any interrupts. The IRQ line needs a pull-up resistor set.

Be also careful with the initialization regarding:

- The speed of GPIO changes (fast)
- The type of output: push-pull
- The signal level

The following table provides an overview over the GPIO configurations:

	CLK	MOSI	MISO	CS	IRQ
Direction	Output	Output	Output	Output	Input
Drive Mode	Push-Pull	Push-Pull	Push-Pull	Push-Pull	None
Pull-Up	Not required	Not required	No	Not required	Pull Up
Speed	Fast	Fast	Fast	Slow	Fast
Mode for SPI	SPI	SPI	SPI	SPI or GPIO	GPIO
Mode for GPIO	GPIO	GPIO	GPIO	GPIO	GPIO

4.5.1.2.2 SPI Mode The device works with CPOL=1 (clock polarity) and CPHA=1 (clock phase), meaning that the clock is pulled high in idle state and the data should be read on the rising clock edge. This is also frequently referred to SPI mode 3. This must be configured accordingly:

- CPOL = 1
- CPHA = 1

4.5.1.2.3 SPI Speed The speed is crucial to achieve a high frame rate when performing continuous measurements: As each measurement requires around 500 bytes of data, the data transfer alone will take 4000 times the SPI clock period, not including the measurement itself and the preparation of the SPI transfers.

It is therefore recommended to choose a high transfer speed that is still compatible with the electrical requirements of the sensor as given in the data sheet and the environment in which it is built. An SPI clock frequency in the range of 10 to 12 MHz is a good starting point.

Note

It may not be easy or even possible to set any SPI clock speed. Instead, the SPI speed is often directly coupled with the system clock via a divider.

4.5.1.2.4 DMA For the DMA (direct memory access) transfer, usually two seperate channels need to be set up for the data read and write.

As the kind of operation always has to be specified by the master within the transfer, the SPI transfers have only two modes:

- Transmit only
- Transmit and receive

Both kinds of transfer are started via the S2PI_TransferFrame() function. While the `txData` pointer always points to valid data, the `rxData` pointer is set to `0` if no data shall be transmitted.

Note

rxData and txData frequently point to the same address. While this should usually be no problem, as the memory byte needs to be read before and stored after the transmission, make sure that the DMA implementation of your MCU supports this. Otherwise, the receive data should be transferred to a temporary buffer and copied to the destination after reception.

4.5.1.2.5 DMA Interrupts and Callback Function Usually, the SPI transmission requires a callback function to be triggered after the transmission is complete.

Typically, in the DMA setup, the device provides DMA complete interrupts that can call the callback function. Also, if the CS signal is applied as GPIO (soft CS), it can be unasserted here.

However, this DMA complete callback function may already start the next SPI transfer immediately. So all cleanups required after the current SPI transfer needs to be performed before that. This may be complicated as two DMA channels may be involved in the transmit and receive case and perform cleanup on their individual channel only. The callback function must be triggered only after both channels are freed and cleaned up. Also, the CS must be unasserted before the callback function is called.

On the other hand, in the transmit only case, make sure that the DMA interrupt does not call the callback function, or unassert CS, before the last byte is fully transmitted, especially if the SPI speed is slow.

4.5.1.2.6 DMA Interrupt Priority To make use of stable and high frame rates, the SPI interrupt should not be blocked by other possibly longer running interrupts, so the interrupt priority should be chosen sufficiently high. On the other hand, if the target application uses other interrupts for very time sensitive purposes, they should have higher priority, as the callback function may include preparing a new SPI transfer and therefore may take multiple microseconds to return.

See the S2PI module documentation for more details on the SPI interface.

4.5.2 Timer Layer

The Timer Interface implements two timers: a lifetime counter (LTC) for time measurement duties and a periodic interrupt timer (PIT) for the triggering of measurements on a time based schedule.

Warning

> The lifetime counter is mandatory in order to heed the eye-safety limits. Note that this timer must be setup carefully in order to guarantee the laser safety to be within *Class 1*.

4.5.2.1 Lifetime Counter (LTC)

The lifetime counter should be set up to deliver the current time in microseconds. The timer resolution must be in the magnitude of 10 to 100 microseconds. This means, however, that the systick counter driven by the systick interrupt cannot be used, as it typically provides an accuracy in the range of milliseconds.

Basically a hardware and a software approach can be chosen to implement the lifetime counter functionality. In any case, a first timer with at least 16-bit width is set to count the sub-seconds value and the prescaler is preferably set such that the timer wraps around after exactly 1 second. In case of a 16-bit counter the maximum achievable granularity is approximately 15 (1/65336) microseconds which is suitable for most applications.

In a hardware based scenario, a second timer with 32-bits is chained to increase its value whenever the first counter wraps around and thus counting seconds. In a software based scenario, the first timer triggers an interrupt upon the wrap around. A 32-bit software counter representing the seconds is increased within the interrupt service routine.

4.5.2.2 Periodic Interrupt Timer (PIT)

Note that the periodic interrupt timer is mandatory only if the user requires the measurements to be started autonomously in the background on a time based schedule. Simple implementations may trigger measurements on demand from the foreground thread and thus the PIT is not required. A weak implementation is provided within the library so that it is not required to implement the interface if the PIT is not used.

A different timer than the timer for the lifetime counter should be used. Usually, the maximum period that will be used can be reached by a combination of the timer reload value and the prescaler even on a 16-bit timer. Extending the period with software is also possible. This can be done by using an additional software counter that counts up every time the interrupts occurs and invokes the callback only after a given number has been reached.

The granularity of the PIT is highly dependent on the required measurement frame rate. E.g. to achieve 1000 frames per seconds, the PIT must be able to trigger every millisecond!

4.5.3 Interrupt Layer

As described in the sections about the S2PI and Timer layers, the Argus API uses three different kinds of interrupts associated with callback functions that have to be set up during the module initialization:

1. SPI DMA Complete Interrupt: This interrupt signals the completion of the DMA transfer. The callback usually triggers a new S2PI transfer or provides a ready indication to the waiting thread. It occurs multiple times during a measurement.
2. GPIO Interrupt for Data Ready: A low indication on the IRQ line indicates that the requested data was collected on the device and is ready for transfer. This occurs usually twice per measurement (sensor and auxiliary data) and sets up the SPI transfer of the gathered data.
3. Periodic Interrupt Timer: Periodic measurements are triggered from this interrupt. The callback initiates a new measurement, and therefore should happen only once per measurement cycle. If, however, there is no space to hold the measurement results as the previous results were not yet processed, no new measurement is started and the measurement cycle is delayed or skipped.

4.5.3.1 Interrupt Priority

All of these interrupts with their callbacks typically take several microseconds to complete. The callback functions within the API are designed to not induce considerable delays and return as fast as possible.

If other interrupts within the microcontroller are used, their priority should be chosen in the following way:

- The interrupts described above should get high to medium priority in the order described.
- Other interrupts that are very important or time critical below milliseconds should get a higher interrupt priority (typically a lower value).
- Other interrupts that are not as important or less time critical should get a lower interrupt priority (typically a higher value).

4.5.3.2 Concurrency and Interrupt Locking

The callbacks from the interrupts above provide information or trigger new SPI transfers. In order to prevent concurrency issues, the program must be able to lock the occurrence of these interrupts in atomic or critical sections of the code. Therefore, the IRQ Interface shall be implemented.

In general, the IRQs are only locked for very short time (sub-microseconds) in order read the status and update it appropriately without the interruption from a higher priority thread. However a nested implementation of the locking mechanism is required for the library core to work correctly.

Refer to the module documentation to see an example implementation that locks all maskable interrupts. If other critical interrupts are present, an alternative implementation can selectively lock only the interrupts used by the Argus API.

4.5.4 NVM Layer

The Non-Volatile Memory (NVM) layer is an optional interface that provides access to a non-volatile memory hardware, e.g. flash. This is used to permanently save calibration parameters that can be gained by executing the corresponding calibration sequences from the Calibration module. Also user set calibration parameters can be saved into the NVM in order to be available after a system reset or power cycle. If custom calibration parameters does not need to be save within the API, the implementation of the interface can be skipped. A weak implementation is provided in the core library that will disable the NVM feature.

Note

Custom calibration data can for example be crosstalk vectors to compensate the impact of coverglass.

The storage of user defined calibration (and configuration) data can also be achieved by using the corresponding API functions and apply the previous parameters after the system reset manually.

4.5.5 Log Layer

In order to send debug and error messages, a *printf*-like function from the Debug Interface can be implemented. If not required, the implementation can be left out and and dummy default implementation will be used that does not send the error messages anywhere. However, implementing this interface may have an impact on the measurement performance, especially when it is slow (e. g. synchronous transfer over UART).

Note that errors are propagated using the status_t enumeration of status and error codes. Any method within the API returns an error code that gives a hint on the execution status of the routine.

4.6 Step-by-step porting guide

The following step-by-step guide leads through the basic process on getting the API running on any Cortex-M0 based development environment. The steps are demonstrated using the MCUXpresso-IDE and the NXP FRDM-KL46Z development platform from NXP that comes also with the AFBR-S50 evaluation kit. It should be an easy task for experienced embedded software developer to follow the steps on its dedicated development environment.

Note

This is a brief guide that shows the basic approach for porting the API only. An extensive porting guide to a Cortex-M4 architecture is available on the Broadcom website. This document shows the full task of porting the *AFBR-S50 API* to a new processor platform on the example of a STM32F403RE Cortex-M4 microprocessor.

4.6.1 Create a new project in your environment

The first step would be to create a new empty project or use an existing one. Usually this is done utilizing the provided platform specific SDK provided by the vendor. In case of the FRDM-KL46Z, the MCUXpresso-IDE and the MCUXpresso-SDK is used to create a project with at least SPI (with DMA mode), GPIO and PIT (periodic interrupt timer) support. The new project is tested using the "Hello World" print statement.

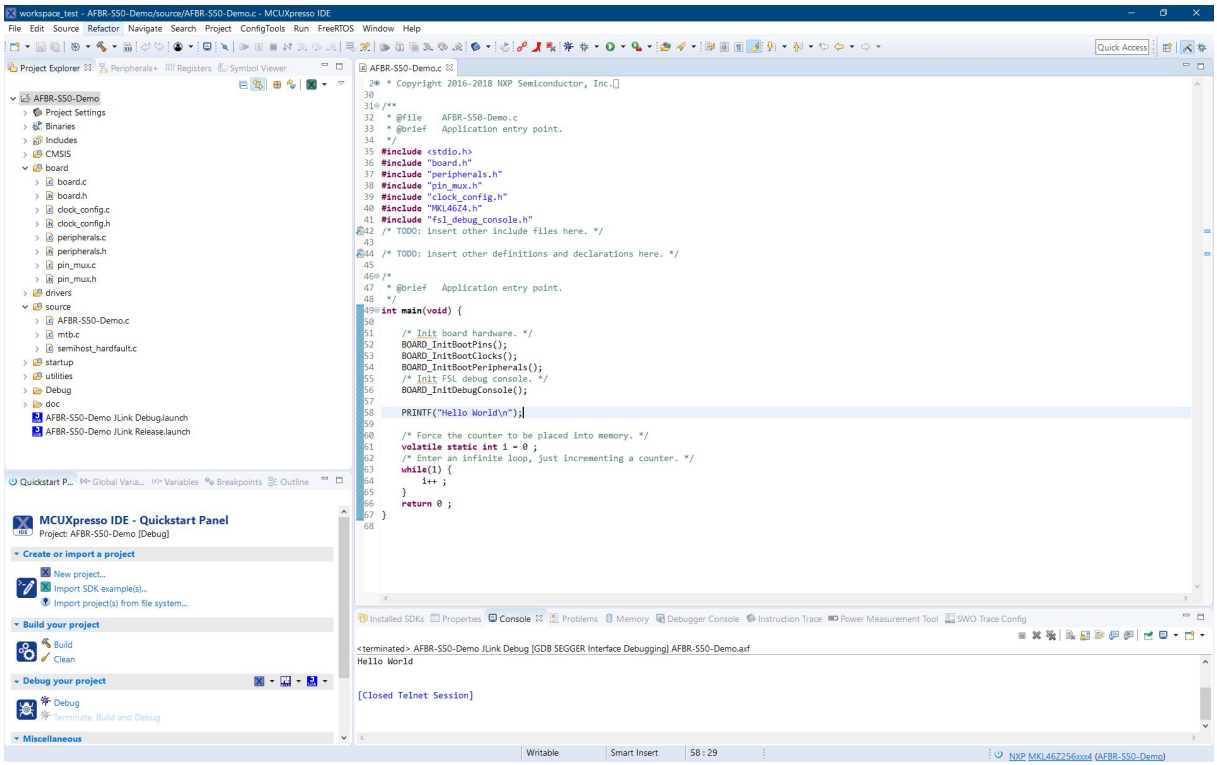


Fig. 4.2: Creating a new SDK project.

4.6.2 Implement hardware interfaces

After the successfully creating, building and testing the new SDK project, the hardware interfaces required by the AFBR-S50 Core Library need to implemented. Basically there are two approaches: either forward the commands to the provided SDK functionality from the MCU vendor or implement an individual version from scratch.

In order to start, the include files must be referenced in the project and a source file is required for each interface file. The simplest way to include the API into your project is to copy the files. Go to the install directory if the AFBR-S50 SDK and find the files in *[INSTALL_DIR]\Device\Lib* (see Fig. 4.3).

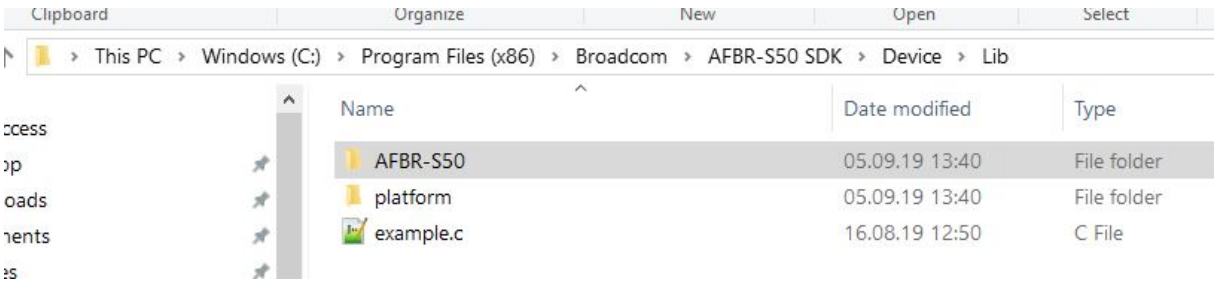


Fig. 4.3: The AFBR-S50 library directory.

The *AFBR-S50* library and include files are in the AFBR-S50 folder which needs to be copied into the project.

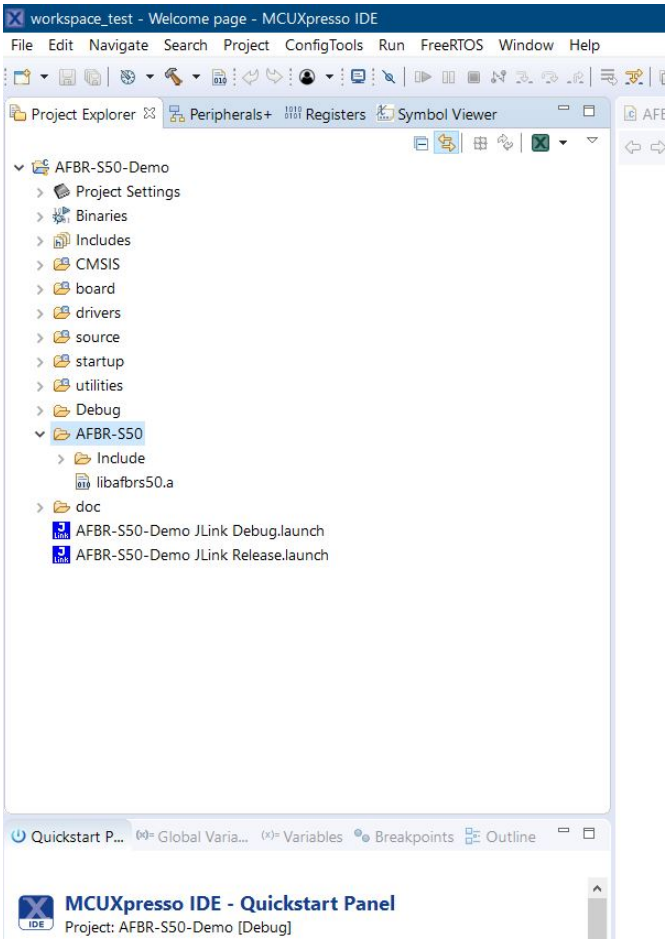


Fig. 4.4: The AFBR-S50 library files are copied into the project folder.

Note: The example.c file as well as the platform folder belonging to the examples. See the Getting Started Guide for more information. The platform contains reference implementation of the platform layer which can also be referred for customers implementations.

The include path to the AFBR-S50 header (*AFBR-S50\Include*) files must be added to the project setting. Afterwards and source file for each individual header file in the *AFBR-S50\Include\platform* folder must be created and filled with definitions for each function declaration in the header files. Therefore the following files are created:

- *argus_s2pi.c*
- *argus_log.c*
- *argus_timer.c*
- *argus_irq.c*

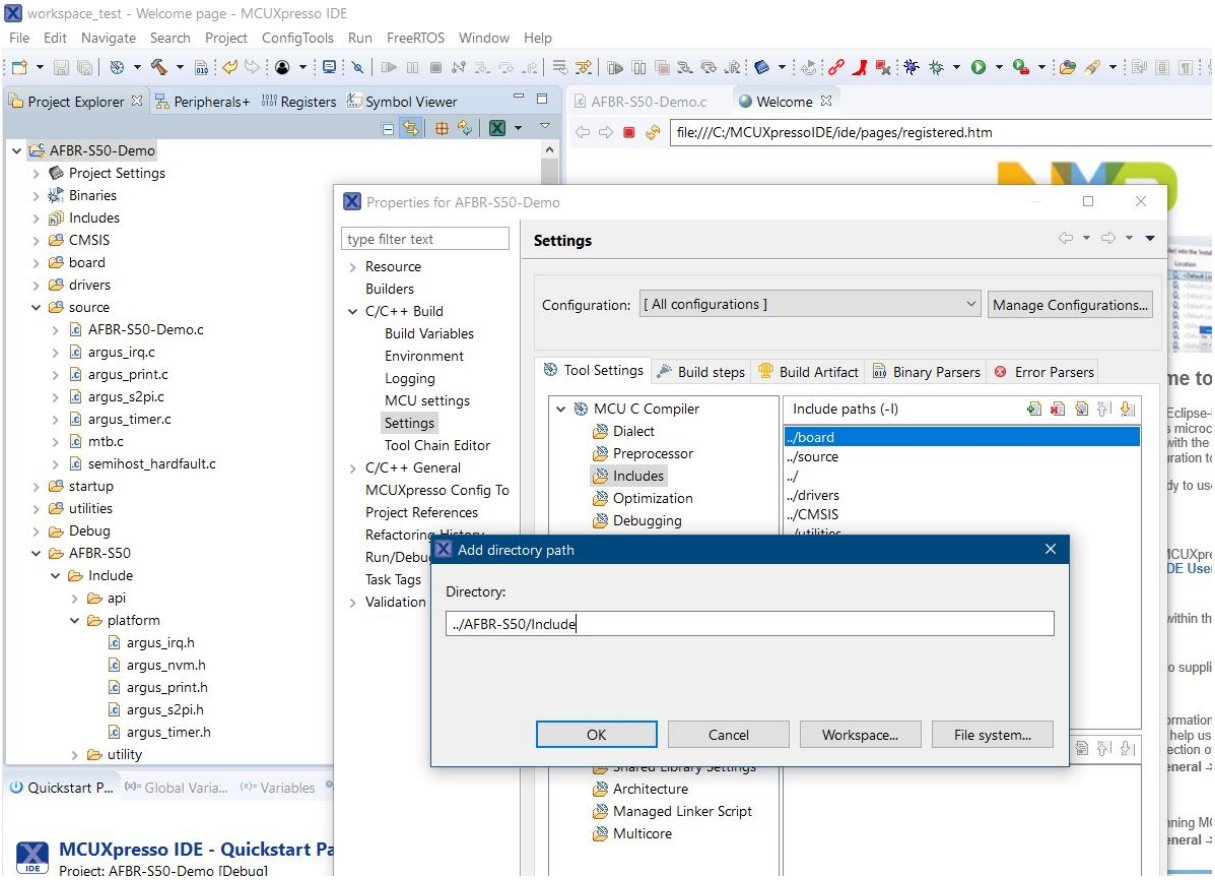


Fig. 4.5: Setup the include path and add the ∗AFBR-S50\Include∗ folder. Afterwards create a source ∗.c file for each header ∗.h file in the ∗AFBR-S50\Include\platform∗ folder. Create an empty function body for each function declaration in the corresponding header files.

Note that the *argus_nvm.h* is ignored in this examples since the usage of the NVM is not required for this demonstration. Default implementations in the AFBR-S50 library core will be used that do not use any non-volatile memory module.

For example, the *argus_timer.c* file could look like this with empty function bodies:

```
#include "platform/argus_timer.h"
void Timer_GetCounterValue(uint32_t * hct, uint32_t * lct)
{ }
void Timer_SetCallback(timer_cb_t f)
{ }
void Timer_SetInterval(uint32_t dt_microseconds, void * param)
{ }
void Timer_Start(uint32_t dt_microseconds, void * param)
{ }
void Timer_Stop(void * param)
{ }
```

After testing the build, implement the actual function bodies. Don't forget to implement and call initialization code as well if required.

Here is a simple example of a timer function:

```
void Timer_GetCounterValue(uint32_t * hct, uint32_t * lct)
{
    IRQ_LOCK();
    LTMR64H = ~(PIT->LTMR64H);
    LTMR64L = (PIT_Freq - 1U) - PIT->LTMR64L;
    IRQ_UNLOCK();
    *hct = LTMR64H;
    *lct = LTMR64L / PIT_ClocksPerUSec;
}
```

Please refer to the platform interface module documentation on details on how to correctly implement the individual layer. Also refer the example implementations of the platform layers that come with the AFBR-S50 SDK, found in *[INSTALL_DIR]\Device\Lib\platform\driver*. These are:

- *irq.c*: implements a basic version of the interrupt locking mechanism that allows nested locking which is declared in argus_irq.h.
- *s2pi.c*: implements the combined SPI and GPIO interface declared in argus_s2pi.h.
- *timer.c*: implements the timer functionality declared in argus_timer.h.
- *uart.c*: contains the function definition for the argus_log.h header and implements a print functionality over an UART interface.

4.6.3 Link Library File

Now that the platform layers are implemented, the library needs to be linked into the project. Therefore add the *AFBR-S50* folder (*[INSTALL_DIR]\Device\Lib\AFBR-S50*) to the library search path and the *libafbrs50.a* file to the linker libraries (i.e. *afbrs50*, leaving away the "lib" and ".a" in case of GNU toolchain).

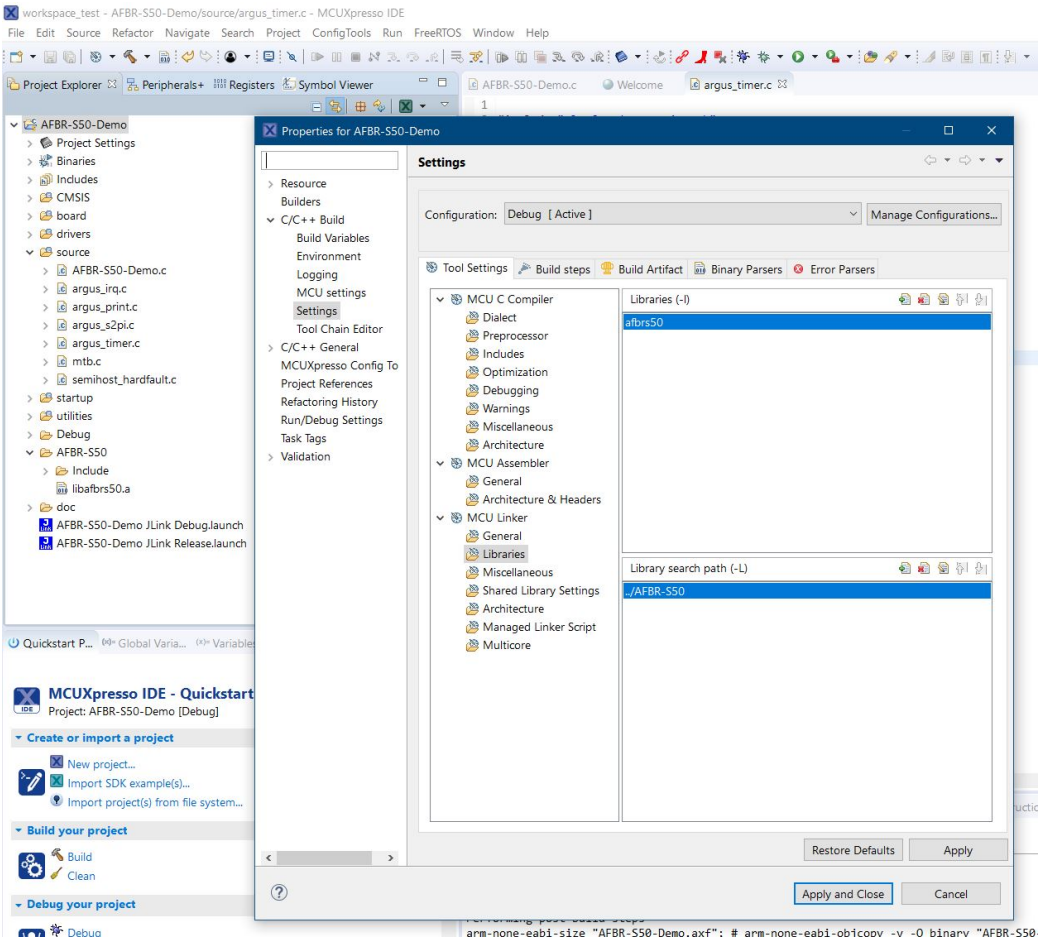


Fig. 4.6: Setup the linker by adding the ∗AFBR-S50∗ library search path and the ∗afbrs50∗ library.

4.6.4 Utilize the API

Now, the *AFBR-S50 API* is ready to use. Include the *argus.h* header and start coding your AFBR-S50 application. Refer to the Getting Started Guide to see an example implementation with basic measurements.

Chapter 5

Explorer App (API Demo)

The **Evaluation Kit** runs with a simple demo application, the **ExplorerApp**. It establishes an USB connection to the **AFBR-S50 Explorer GUI** that is running on a Windows PC. The ExplorerApp provides a simple interface that allows the GUI to transfer configuration and calibration parameters and receive measurement data.

The ExplorerApp hosts the **AFBR-S50 Core Library** on the one hand and implements a **Serial Communication Interface (SCI)** on the other hand. The SCI is an communication protocol that can be used with almost every serial communication interface like UART or USB. The ExplorerApp is the communication slave, while an external host, e.g. the AFBR-S50 Explorer GUI running on Windows, is implemented as a master.

The Evaluation Kit (containing the `NXP Kinetis MKL46z` MCU) uses the USB interface to connect to the external host (e.g. the AFBR-S50 Explorer). Another reference solution is provided, based on the `NXP Kinetis MKL17z` MCU, that uses an UART interface to establish a connection to an external host.

The SCI of the ExplorerApp contains a API that is equivalent to the AFBR-S50 API, but accessible via the serial peripheral hardware. The following chapter gives an overview about the architecture and implementation of the SCI module in the ExplorerApp. After gaining a basic understanding of the implementation, it should be an easy task to adopt the *ExplorerApp* along with the interface to the user requirements and create the host interface that can connect to the provided SCI.

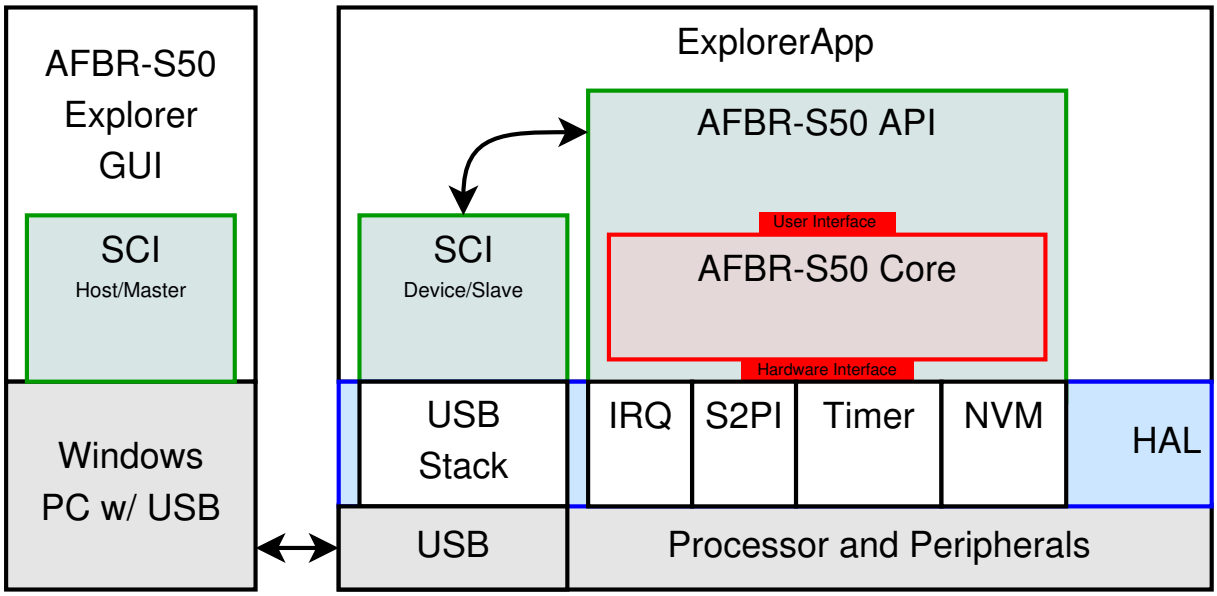


Fig. 5.1: An overview of the ExplorerApp Architecture. The ExplorerApp is hosting the AFBR-S50 API on the one hand and a serial communication interface on the other hand to connect to an external host via serial peripheral like USB or UART. The external host would be a PC with running the AFBR-S50 Explorer GUI in case of the evaluation kit.

5.1 Build And Run the ExplorerApp using MCUXpresso

In order to run the provided ExplorerApp project using the MCUXpressIDE by NXP, execute the following steps. Please also refer to the getting started guide by NXP in case of any trouble: https://www.nxp.com/docs/en/user-guide/MCUXSDKGSUG.pdf

- Download and install the MCUXpresso IDE (recommended v11.1):
  - Go to https://www.nxp.com/design/software/development-software/mcuxpresso-software:MCUXpresso-IDE
  - Click on *Download* and register or sign in to download the installer.
  - Install the IDE.
  - Go to https://mcuxpresso.nxp.com for more information.
- Download and import the KL46z SDK into the MCUXpresso IDE v11.1
  - Open MCUXpressIDE, accept the workspace settings by clicking on "Launch".
  - Click on "Download and Install SDKs" on the "Welcome Page".
  - Go to the "Processors" tab and type "MKL46" into the filter field, select the "SDK_2.x_MKL46Z256xxx4" SDK and click "Install".
  - Accepot the licenses and click on "Finish".
  - After the installation has finished, close the "Welcome" view.

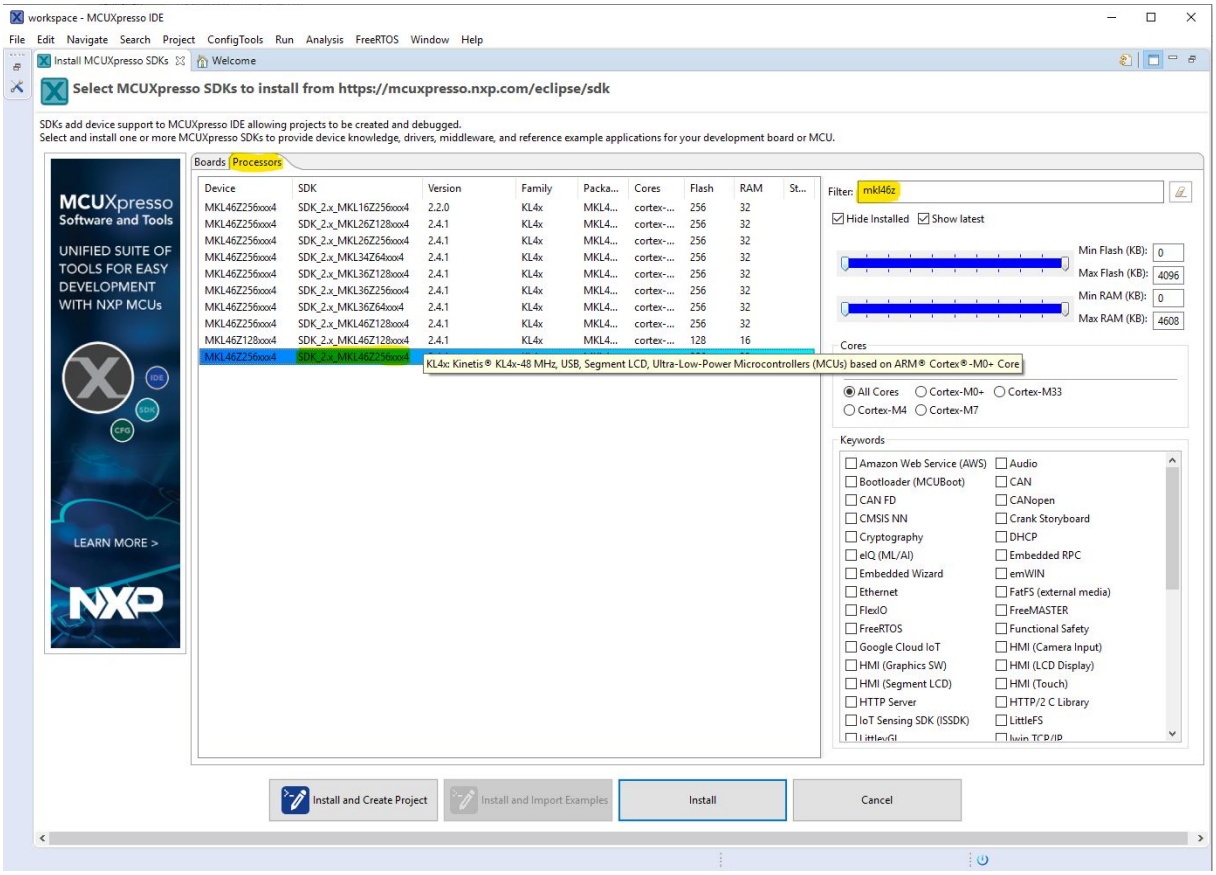


Fig. 5.2: Download and install the SDK files into the MCUXpresso IDE.

- Import the project archive files:

– Go to MCUXpresso IDE -> Quickstart Panel

– Click on "Import projects(s) from file system..."

– Click on "Browse..." in the Archive section

– Browse to "[INSTALL_DIR]\Device\Projects\" (default is "C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Projects\"

– Select the required project archive ("∗AFBR_S50_ExplorerApp_KL46z.zip∗") and click "Open"

– Click on "Next" -> "Finish"

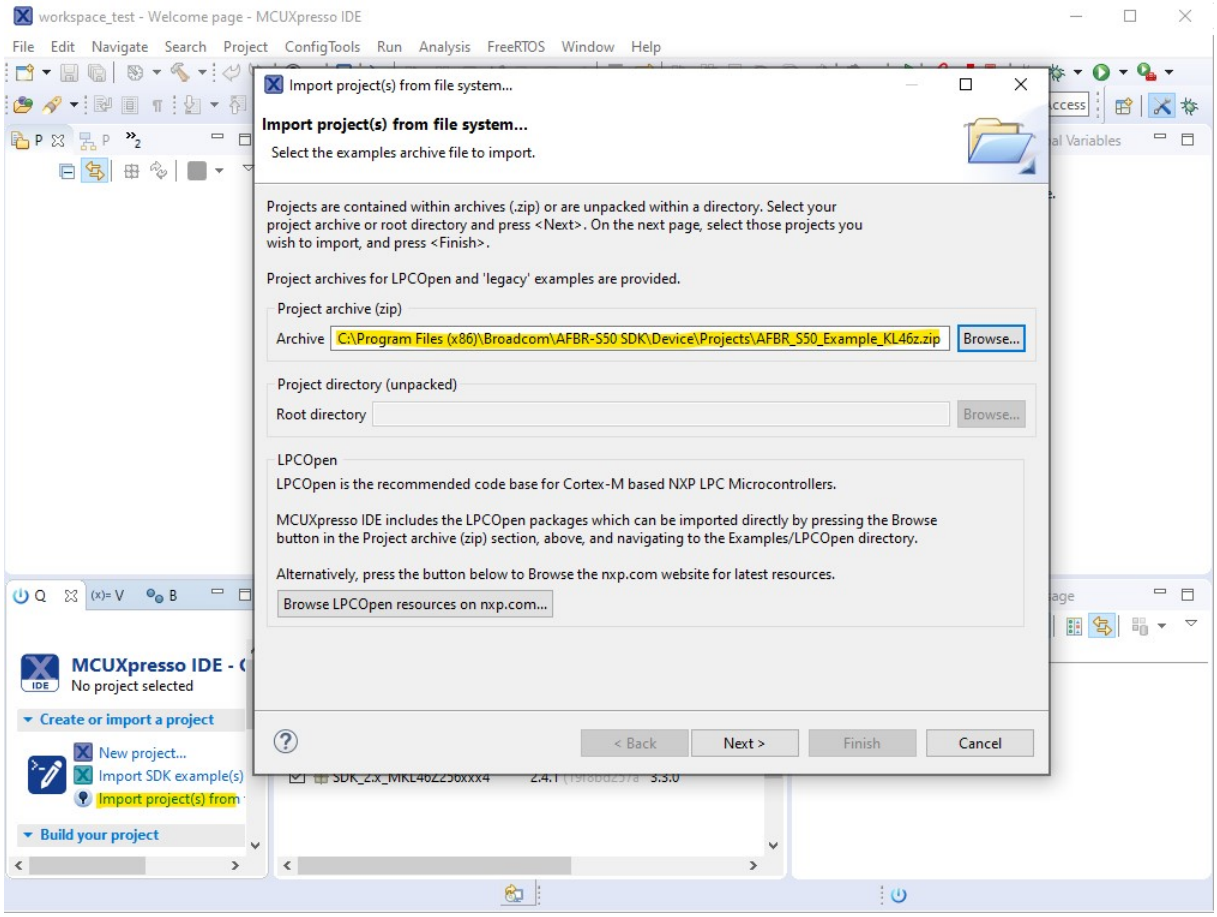


Fig. 5.3: Import the project archive into the MCUXpresso IDE.

- Build the projects:

– Go to MCUXpresso IDE -> Quickstart Panel

– Click on "Build"

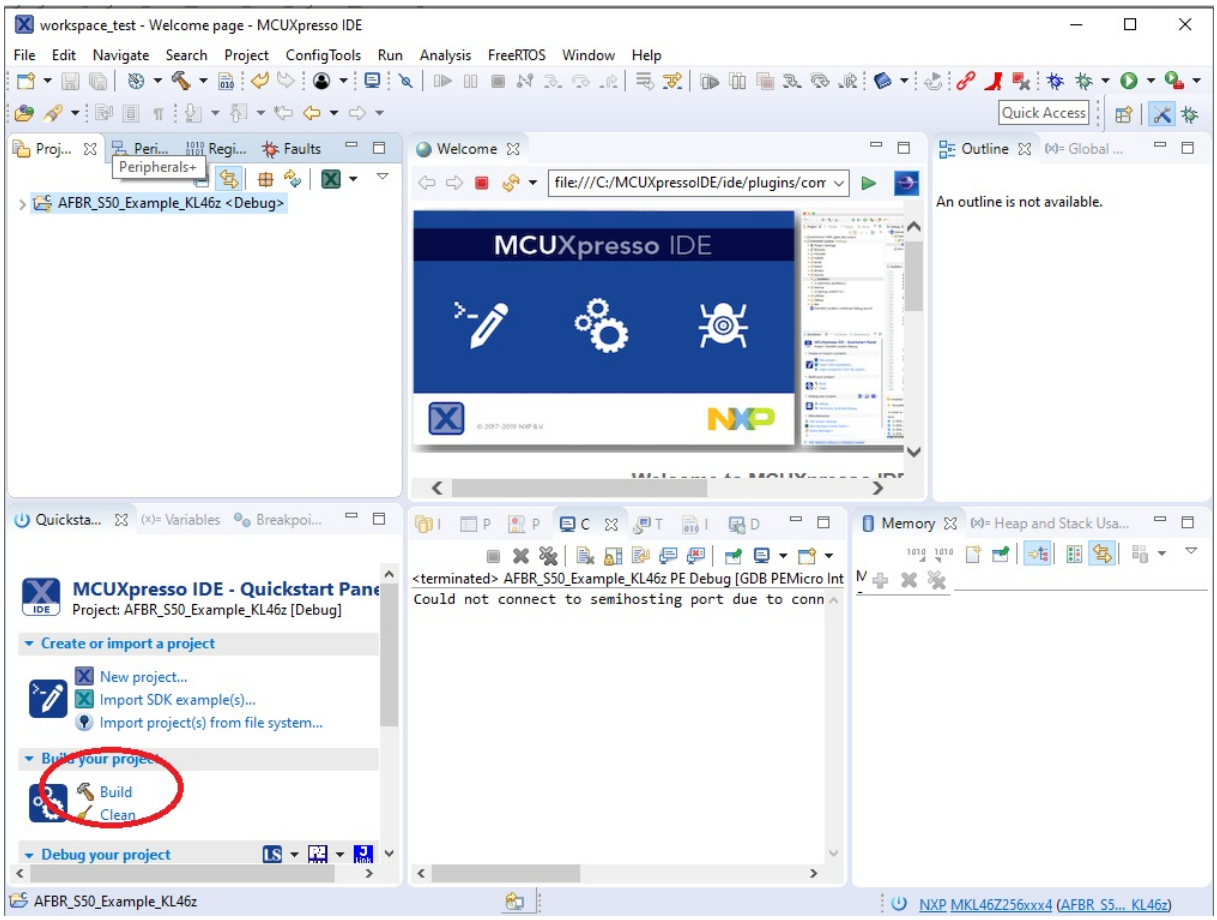


Fig. 5.4: Build the project.

- Download and install the OpenSDA drivers:
  - Go to http://www.pemicro.com/opensda/
  - Download and install *PEDrivers_install.exe*
- Debug and run the project with the OpenSDA debugger:
  - Connect the *OpenSDA* USB port of the KL46z evaluation board.
  - Go to MCUXpresso IDE -> Quickstart Panel
  - Click in the PEMicro Icon (see screenshot).
  - The Debug Probe will be discovered and an according window will show.
  - Click on "OK"

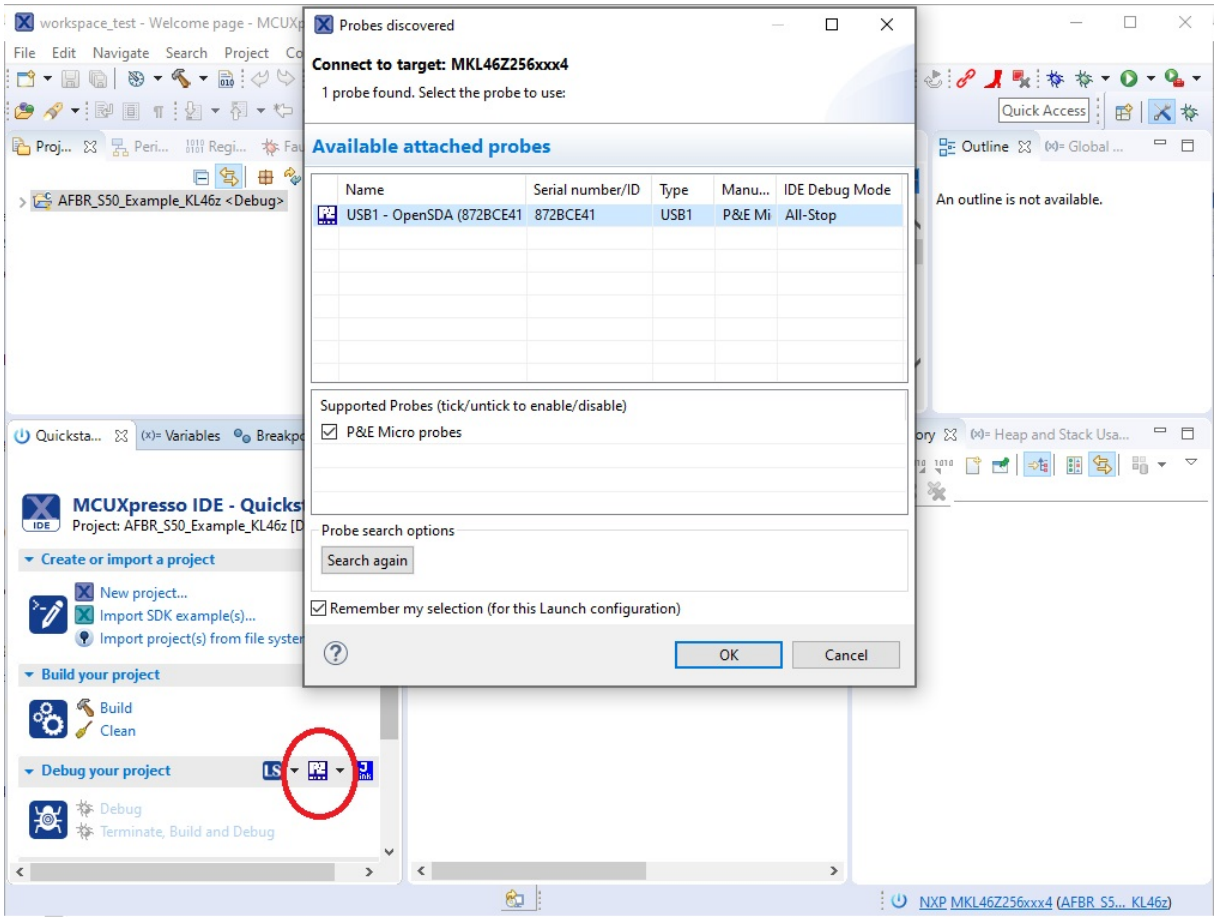


Fig. 5.5: Run and debug the project.

– If the program breaks at the main() function, hit the "Resume" button.

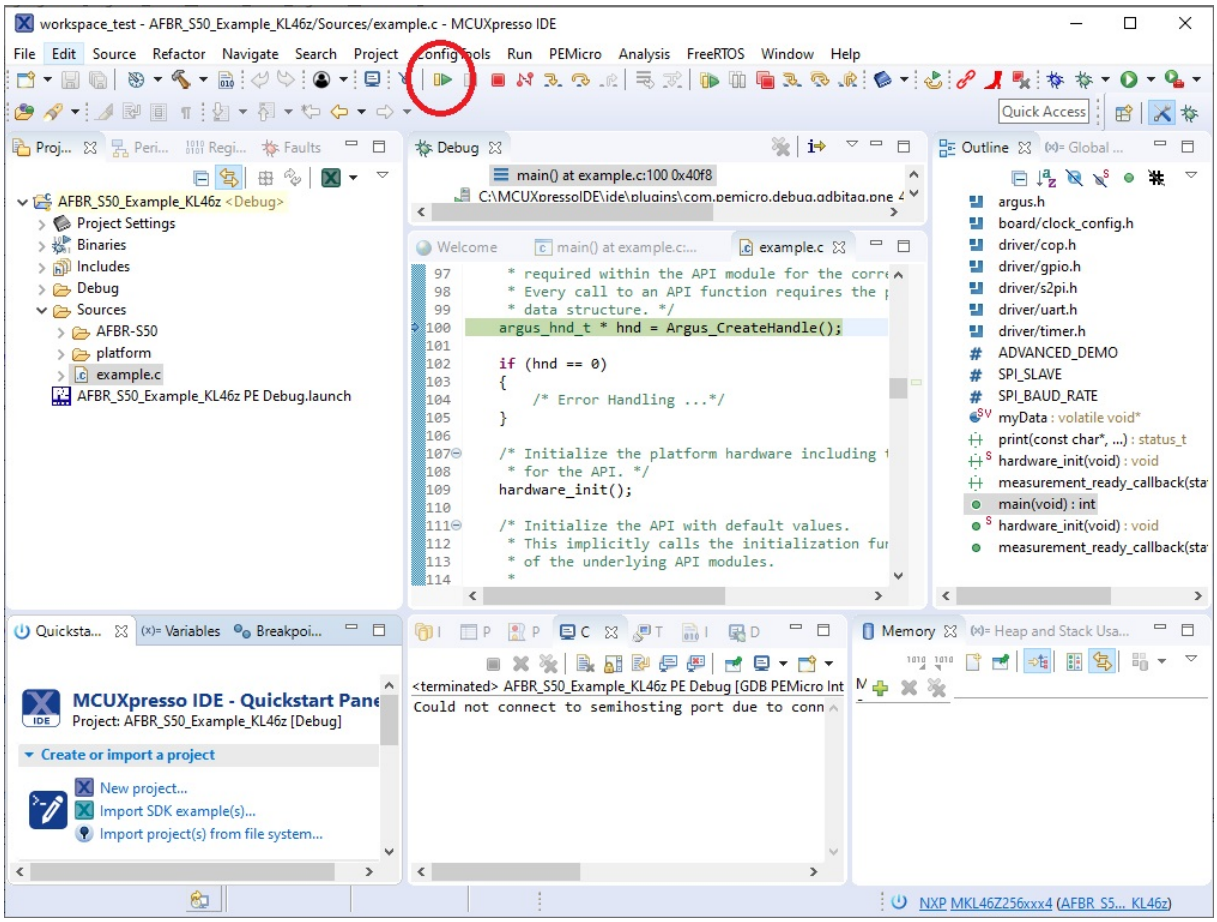


Fig. 5.6: Resume the halted program.

- Connect to the device via USB/UART with your SCI master.

The evaluation kit is build on the FRDM-KL46z Evaluation Kit from NXP. So you may also refer https://www.nxp.com/frdm-kl46z for further information.

5.2 Serial Communication Interface

5.2.1 Introduction

The basic idea is to define simple commands (= 7-bit values) to access the AFBR-S50 API. These commands are either transactions with or without data phase. Usually, the commands with data phase are simple getter and setter commands, e.g. "get distance" or "set frame rate".

Each data package starts by the identifier byte (= command byte), followed by a predefined number of data bytes which may be zero. Finally, the package finishes with a CRC byte to verify the data integrity. These byte sequences are called data frames.

The data frames are transmitted in different manner dependent on the underlying hardware interface. However, at a higher level, the command or message layer does not depend on the hardware interface. The communication happens between various systems, whereby one of them needs to be the master. All other participants are slaves which are controlled by the master. Depending on the underlying hardware, there can be a single (e.g. UART) or

multiple (e.g. SPI or I2C) slaves. In case of the latter, the slaves are addressed by the master via the corresponding hardware architecture.

Each command to a slave is acknowledged after successful execution. If any error occurs, a not-acknowledge is invoked by the slave. Only a single command can be sent to the slave at once, i.e. the slave has to (not-)acknowledge the command before the master can send another one. A timeout can be implemented in the master to check if the slave responds within a given time and is still alive or if it is stuck in some invalid state.

5.2.2 Architecture

The architecture of the SCI (see Fig. 5.7) consists of several layer. Each is communicating on a specific level with its corresponding counterpart.

The lowermost layer is the hardware layer which is the hardware abstraction layer for the underlying hardware, e.g. UART or I2C. The hardware layer transfers data bits over a physical link.

The second layer is the data link layer. It takes care about a reliable transmission of data frames, i.e. a bunch of associated bytes. Therefore it is responsible for putting together data by framing it with a start and a stop byte. It also applies the byte stuffing and finally adds an CRC value to allow the detection of invalid frames.

The third layer is the protocol or message layer. Its responsibility is to transfer messages or commands via the data link layer. In order to achieve a reliable connection, the handshaking is also implemented into this layer via acknowledgment messages.

The last and uppermost layer is the Application or API layer. It provides high level functionality to transfer application specific data.

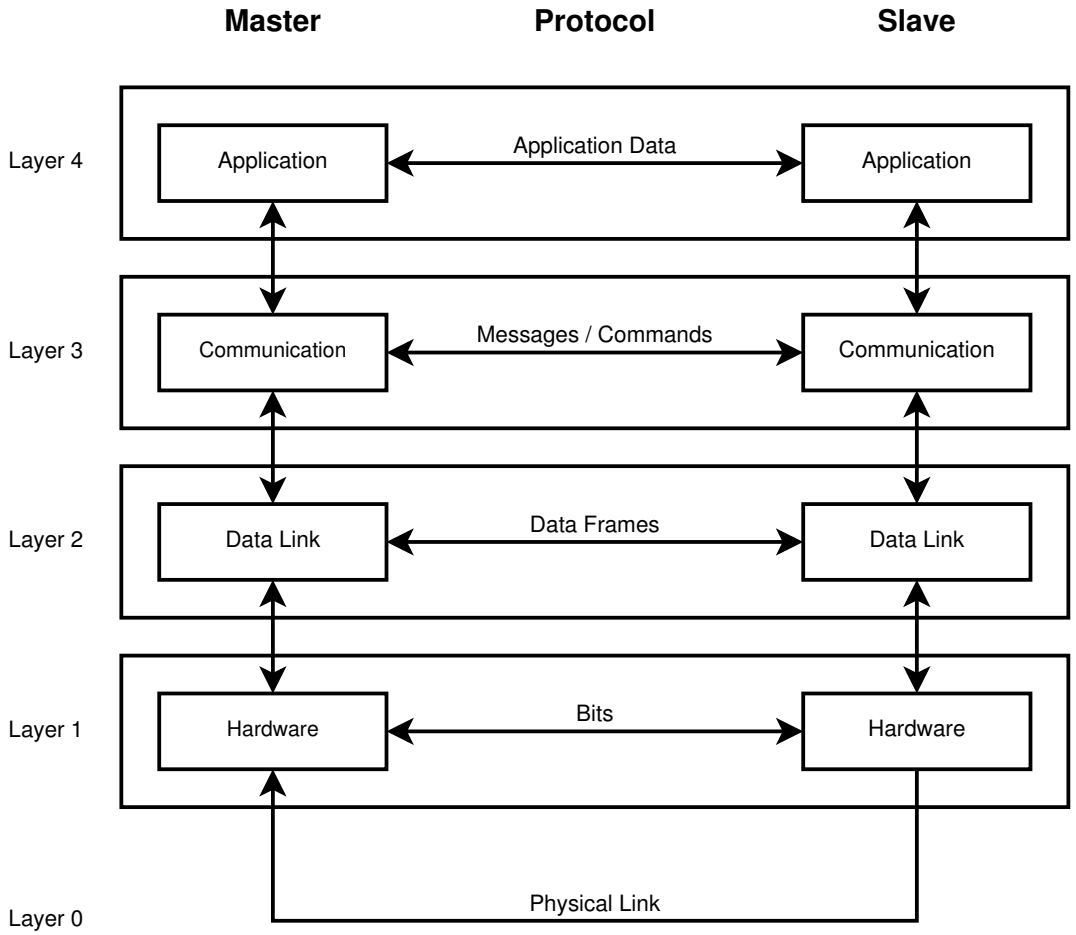


Fig. 5.7: The SCI Layer Model.

To sum up:

1. Hardware Layer: transfer bits over physical medium
2. Data Link Layer: data framing, byte stuffing and CRC
3. Message Layer: command, messages and handshaking
4. Application Layer: host application specific data

See also

See also the OSI model, which was used as an reference: https://en.wikipedia.org/wiki/OSI_model

5.2.3 Hardware Layer

5.2.3.1 UART

The UART interface support only point-to-point communications. It has an independent line for sending as well as receiving data and the slave can transmit data at any time without special actions from the master are required. Therefore, this mode does not require an interrupt line to inform the master about new data ready or error conditions. The data is just transmitted immediately which means that the master must always listen to its Rx line.

The data framing is realized with byte stuffing. There are three special bytes, the start, stop and escape bytes, that are used to determine the boundaries of a data frame. In order to make the start and stop bytes unique and keep the full data range per byte, the corresponding data bytes are inverted and escaped with the escape byte.

In order to provide a handshaking mechanism, the slave acknowledges the successful reception of an data frame (and the successful invocation of the corresponding command) with a short ACK (= acknowledged) or NAK (= not_acknowledged) message within a define time.

Furthermore, due to the independent TX line, the special feature of log and error messages are supported by the UART protocol.

5.2.3.2 SPI

The SPI interface support multiple slave mode via the chip select (CS) lines. Data transfers can only be initiated by the master and thus an extra IRQ line is used to give the slave a chance to call the masters attention to it. This is however optional and the alternative method would be polling the status by the master.

The data framing is realized via the CS. After the command byte, the data is transferred either by on the MISO or MOSI for a read or write command respectively.

The handshaking is implemented via the IRQ line. If an error occurs, the IRQ is pulled to low. The master can now read the corresponding status in order to get the root cause of the IRQ. Also the new measurement ready event is determined via the IRQ. In addition, the acknowledgment of the successful reception and execution of a command could be implemented via the interrupt; the master would responsible for reading the acknowledge status.

5.2.3.3 I2C

The I2C interface supports multiple slave mode via the device address bytes. Data transfers can only be initiated by the master and thus an extra IRQ line is used to give the slave a chance to call the masters attention to it. This is however optional and the alternative method would be polling the status by the master.

The data framing is realized via the usual I2C protocol. Every frame is started with the I2C start condition followed by the devices write address and then master writes the command. In case of a write command, the data can follow immediately. In case of an read command, the devices read address is placed after an repeated start condition. The slave will put its data to the SDA line afterwards.

Besides the already build-in acknowledgment mechanism form the I2C protocol, the reception of an invalid data frame is advertised via the IRQ line. If an error occurs, the IRQ is pulled to low. The master can now read the corresponding status in order to get the root cause of the IRQ. Also the new measurement ready event is determined via the IRQ. In addition, the In addition, the acknowledgment of the successful reception and execution of a command could be implemented via the interrupt; the master would responsible for reading the acknowledge status.

5.2.4 Command Protocols

5.2.4.1 Master to slave transfer

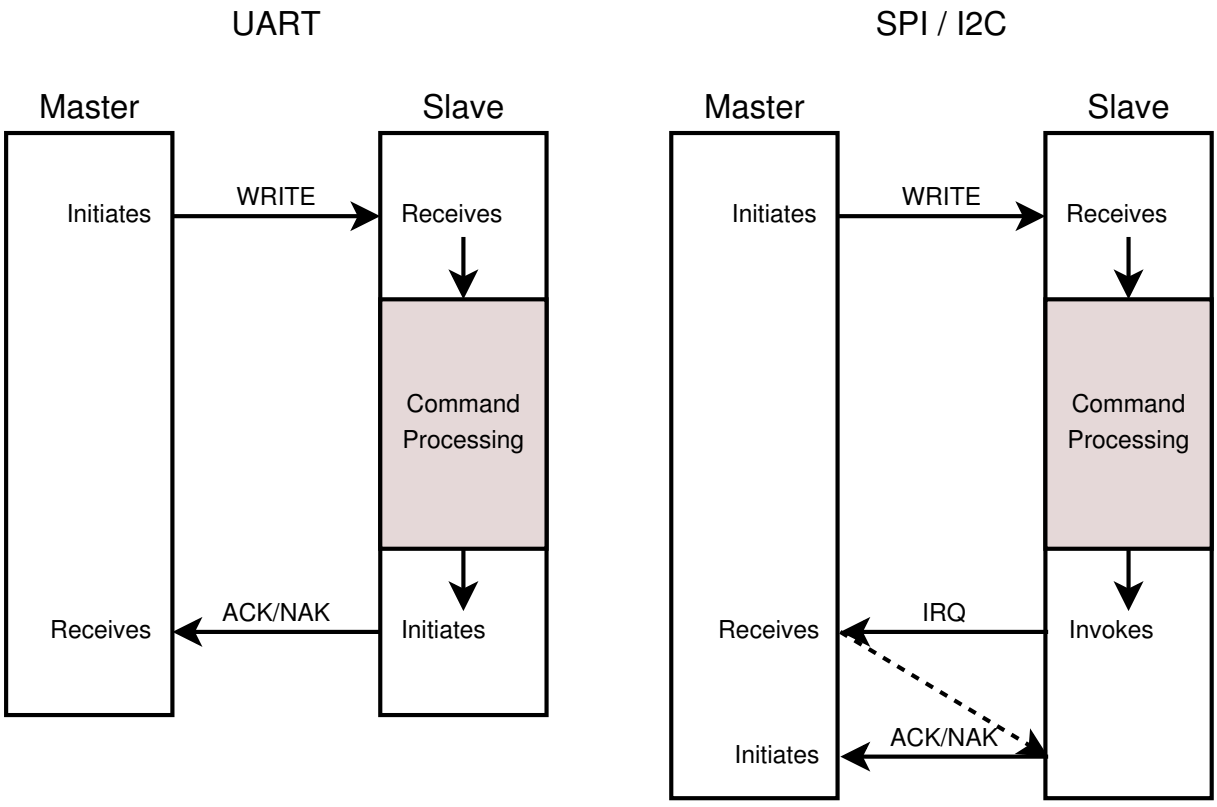


Fig. 5.8: The master to slave communication. The left side shows the UART hardware and the right side the version for SPI/I2C hardware.

In case of UART, the master simply send data via its Tx line. After processing, the slave responds with an acknowledge or not-acknowledge signal on the masters Rx line.

In case of SPI or I2C, the slave can not send data without the master initiating the transfer. Thus, an additional IRQ is used to give the slave the chance to call the masters attention. In case of no IRQ available, the master must poll the slave on a regular basis. So after processing, the slave signals when he is ready to send the acknowledge or not-acknowledge signal and afterwards the master must initiate the transfer from slave to master.

5.2.4.2 Slave to master transfer

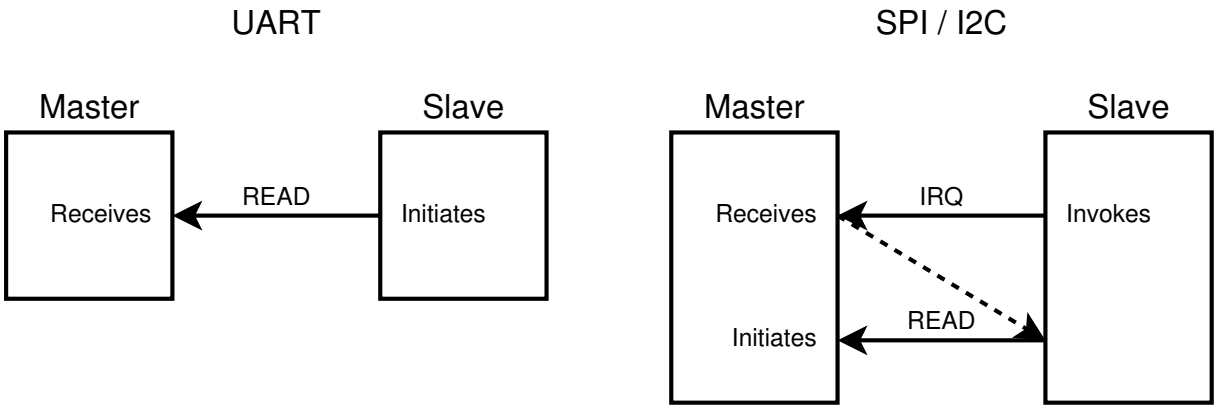


Fig. 5.9: The slave to master communication. The left side shows the UART hardware and the right side the version for SPI/I2C hardware.

In case of UART, the slave simply sends data via its Tx line. After receiving on the master, the sent data is dismissed. If an error occurs, the master is responsible to react accordingly, e.g. re-initiating the transfer by re-sending the previous command.

In case of SPI or I2C, the slave can not send data without the master initiating the transfer. Thus, an additional IRQ is used to give the slave the chance to call the masters attention. In case of no IRQ available, the master must poll the slave on a regular basis. The slave signals when he wants to send data via the IRQ and afterwards the master must initiate the transfer from slave to master.

5.2.5 Command Byte Format

Every command message is identified by the first byte in a data frame. This byte is an unique number that is mapped to a specified parameter/value/command.

The command identifier consists of 7-bit.

In consideration of future applications (advanced commands, addressed commands, ...), the MSB is reserved (escape bit) and always 0. The remaining 7-bits determine the command.

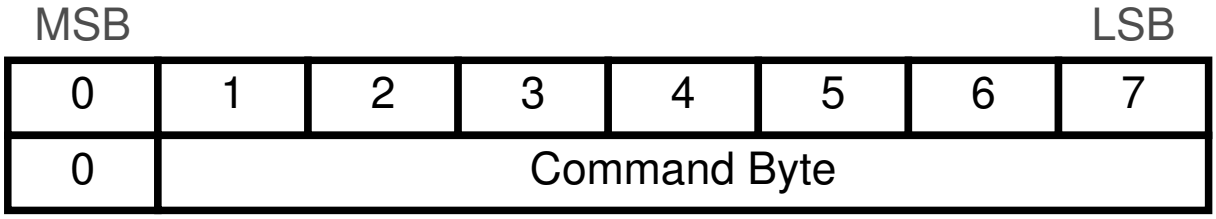


Fig. 5.10: The command byte format.

Reserved Command Bytes

Byte	Comment
$>$ 127	MSB is reserved for later use
0x02	ASCII: Start of text
0x03	ASCII: End of text
0x1B	ASCII: Escape
0x21	ASCII: ! - reserved for later use
0x23	ASCII: # - reserved for later use
0x24	ASCII: $ - reserved for later use
0x3F	ASCII: ? - reserved for later use

5.2.6 Command Types

There are several command types defined:

- **Command (cmd)**: A data frame with a command byte that determines a simple command message that will invoke an action on the slave side. The commands are send from the master to the slave. The slave executes a corresponding function. Usually there is no data phase but in some cases there might be some (optional) function parameter.
- **Setter (set)**: Command byte followed by a given sequence of data bytes representing the data that needs to be transferred from the master to the slave.
- **Getter (get)**: A request from the master to read data from the slave. The actual data read phase depends a bit on the underlying hardware. While for SPI and I2C, the data is read directly in the context of the message, the response is sent as an autonomous message from the slave via its UART Tx line. Note that a get message is actually a command message that invokes the data transfer from the slave to the master in case of UART. Note that for some get messages there might be a short data phase for additional specification of the data to be read, e.g. an index number.
- **Automatic** / **Autonomous Push (Auto** / **Push)**: A message or data transfer that is initiated by the slave. Depending on the hardware, the slave requests a data transfer by the data-ready pin or simply invokes the transfer in an autonomous way. The first is the case for I2C and SPI modes while the latter is the case for UART mode. This type of message is utilized by the slave to send log messages or establish an data stream of new measurement data to the master without the requirement of polling the line.

Thus a single command byte can have up to three different intentions.

- **cmd**: Executing actions.
- **get**: The usual getter for read-only data.
- **set** / **get** The setter/getter combination e.g. for configuration parameters that can be applied to the slave and also read back.
- **auto** / **push**: Data that is only send from the slave as needed without the request form the master, e.g. log messages or data streaming.

5.2.7 Data Frame Format

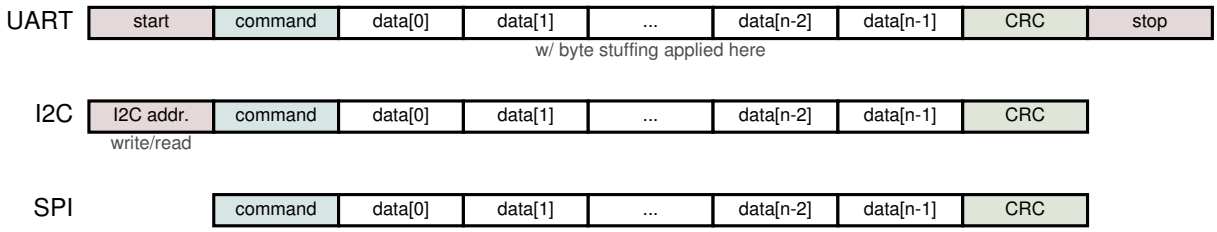


Fig. 5.11: The data frame format for different hardware.

Each message or command that is sent over underlying hardware is put into a data frame. A data frame is a sequence of bytes with variable length that contains arbitrary data. Depending on the hardware, the boundaries of the data frames are given in an unique way.

The first byte within a data frame is the command byte that uniquely determines the purpose and thus the format of the data frame. The last byte within the data frame is the security byte that contains a CRC value that guarantees the data integrity of the receive frame. Between the command and CRC bytes, there is an optional data phase of arbitrary length. The format and interpretation of the data is determined by the command byte in the higher layers of the communication stack and does not have any influence for the data framing.

5.2.7.1 Byte Stuffing Algorithm

In case of SPI and I2C, the data framing is incorporated into the hardware protocol itself, while for the UART interface, a software solution is implemented. The SPI data frame is given by the chip select signal. Each data byte that comes while the chip select line stays at low are combined to a single frame until the chip select is released. The I2C protocol implements an embedded start and stop signal that gives the boundaries of a single data frame.

The UART version implements the data framing in software via byte stuffing, since there is no mechanism to detect the start of a new data package build into the hardware interface. The ideas is to reserve some unique byte values to serve as special control signal, e.g. start and stop bytes. Now, all bytes within a start and a stop byte are interpreted as data bytes for a single data frame. In order to not loose the full range of 256 valid values per data byte, an additional escape signal is introduced. Whenever a byte value equal to the value of start, stop or escape byte appears as a data byte within the current frame, an escape byte is added prior to the byte in question in order to signal that the following value is not an control signal but an ordinary data value. In order to increase security further, escaped data bytes are inverted to make the three control values unique.

Here are the control byte definitions:

Name	Byte	Comment
Start Flag	0x02	ASCII: Start of text
Stop Flag	0x03	ASCII: End of text
Escape Flag	0x1B	ASCII: Escape

See also

https://eli.thegreenplace.net/2009/08/12/framing-in-serial-communications

5.2.7.1.1 Byte Stuffing Algorithm for Sending Data

The algorithm to send a data frame with byte stuffing:

1. Create a new data buffer with start byte included at the first position.
2. Append the data in the buffer (command and data) and add escape bytes on the fly, invert escaped bytes.
3. Calculate the CRC on that buffer, but ignore all escaped bytes, i.e resolve byte stuffing
4. Add the CRC value (and maybe escape signal) and the Stop byte.
5. Send the buffer.

5.2.7.1.2 Byte Stuffing Algorithm for Receiving Data The algorithm to receive data with byte stuffing:

1. After detecting a start byte, start receiving all data until the subsequent stop byte.
2. Write the received bytes into a buffer, removing escape bytes and invert escaped bytes on the fly
3. Calculate and verify the CRC value.
4. Evaluate the data buffer by invoking the corresponding function belonging to the given command byte.

5.2.7.2 Error checking: 8-bit CRC

In order to guarantee the data integrity of the received data frame, a CRC value is added to each data frame.

CRC8_SAE_J1850_ZERO definitions:

Name	Byte
CRC Generator Byte	0x1D
CRC Start Value	0x00

Refer to the following link to verify your implementation of the *CRC8_SAE_J1850_ZERO* algorithm:

See also

http://www.sunshine2k.de/coding/javascript/crc/crc_js.html

5.3 Command Definitions

5.3.1 Overview

Go to: Command Overview

5.3.2 Details

Go to: Command Details

5.4 Python Example on using the SCI interface

Here is an example that sets some configuration parameters (i.e. data output = 1D data only; measurement frame rate = 5 Hz) and starts timer based measurements. The range is extracted from the received data structure and printed to the console.

The example is very basic though. The sent data frames are manually created (i.e. data encoding, byte stuffing and CRC are hard coded). It ignores the received Acknowledge/NotAcknowledge commands and cannot handle unexpected data from the device.

The file runs with Python (3.6) and requires the Python serial module (pySerial).

Please find the file in "[INSTALL_DIR]\Device\Examples\sci_python_example.py" (default is "C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Examples\sci_python_example.py").

```
# Example for using the AFBR-S50 API with UART interface
# ################################################################################
#
#
# Prepare your evaluation kit (w/ NXP MKL46z MCU) by flashing the UART binary
# to the device. Connect the OpenSDA USB port (NOT the one labeled with KL46Z)
# to your computer. Go to the Device/Binary folder install directory of you SDK
# (default: C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Binary) and copy
# the AFBR.S50.ExplorerApp.vX.X.X_KL46z_UART.srec (not the *_USB.*!!) file to
# the OpenSDA USB drive.
#
# After flashing, the device is ready to receive commands via the OpenSDA serial
# port. Go to your device manager to find out which COM port is assigned to the
# device. Type it to the "port" variable below, before starting the script.
#
# Use Python 3 to run the script. The script requires the pySerial module which
# might need to be installed. See: https://pyserial.readthedocs.io/en/latest/index.html
#
#
# The script sends configuration commands to set the data output mode to 1D data
# only and the frame rate to 5 Hz. After setting the configuration, the
# measurements are started and the range is extracted from the received data
# frames and printed to the console.
#
#
# Note: The CRC values are calculated manually and added before the frames are
# sent. You can use the online calculator from the following link w/
# CRC8_SAE_J1850_ZERO to obtain the CRC values for a frame:
# http://www.sunshine2k.de/coding/javascript/crc/crc_js.html
#
# ################################################################################
import serial
# input parameters
port = "COM4"
sample_count = 100
# byte stuffing definitions
start_byte = b'\x02'
stop_byte = b'\x03'
esc_byte = b'\x1B'
def write(tx: bytes):
    print("Sending: " + tx.hex())
    ser.write(tx)
    return
def read():
    # read until next stop byte
    rx = bytearray(ser.read_until(stop_byte))
    # remove escape bytes if any
    rxi = rx.split(esc_byte)
    rx = b''
    for i in range(len(rxi)):
        rxi[i][0] ^= 0xFF # invert byte after escape byte (also inverts start byte, but we don't care..)
    rx = rx.join(rxi)
    # extract command byte (first after start byte)
    cmd = rx[1]
    # interpret commands
    if cmd == 0x0A: # Acknowledge
        print ("Acknowledged Command " + str(rx[2]))
    elif cmd == 0x0B: # Not-Acknowledge
        print ("Not-Acknowledged Command " + str(rx[2]) + " - Error: " + str((rx[3] << 8) + rx[4]))
    elif cmd == 0x06: # Log Message
        print("Device Log: " + str(rx[8:-2]))
    elif cmd == 0x36: # 1D Data Set
        # Extract Range:
        r = (rx[12] << 16) + (rx[13] << 8) + rx[14]
        r = r / 16384.0 # convert from Q9.14
        print ("Range[m]: " + str(r))
```

```
    else: # Unknown or not handled here
        print ("Received Unknown: " + rx.hex())
    return rx
# open serial port w/ "11500,8,N,1", no timeout
print("Open Serial Port " + port)
with serial.Serial(port, 115200) as ser:
    print("Serial Open " + port + ": " + str(ser.is_open))
    # discard old data
    ser.timeout = 0.1
    while len(ser.read(100)) > 0: pass
    ser.timeout = None
    # setting data output mode to 1D data only
    print("setting data output mode to 1d data only")
    write(bytes.fromhex('02 41 07 F5 03'))
    read()
    # setting frame time to 200000 µsec = 0x00030D40 µsec
    # NOTE: the 0x03 must be escaped and inverted (i.e. use 0x1BFC instead of 0x03)
    print("setting frame rate to 5 Hz (i.e. frame time to 0.2 sec)")
    write(bytes.fromhex('02 43 00 1B FC 0D 40 85 03'))
    read()
    # starting measurements
    print("starting measurements in timer based auto mode")
    write(bytes.fromhex('02 11 D0 03'))
    read()
    #read measurement data
    print("read measurement data")
    for i in range(sample_count):
        read()
    # starting measurements
    print("stop measurements")
    write(bytes.fromhex('02 12 F7 03'))
    read()
    ser.close()             # close port
```

5.5 Command Overview

The following section contains the current command overview as implemented in the *ExplorerApp*. The implementation can be found in the "Sources\explorer_app\api" folder of the "AFBR_S50_ExplorerApp_KL46z" project. See also Explorer App (API Demo).

5.5.1 Generic Commands

Caption	Byte	Type	Comment
Invalid Command	0x00	-	Reserved! Invalid Command;
Acknowledge (ACK)	0x0A	auto/push	Slave does acknowledge the successful reception of the last command.
Not Acknowledge (NAK)	0x0B	auto/push	Slave does not-acknowledge the successful reception of the last command.
Log Message	0x06	auto/push	An event/debug log message sent from the slave to inform the user.
Test Message	0x04	set / get	Sending a test message to the slave that will be echoed in order to test the interface. The slave will echo the exact message including the CRC values from the original message.
MCU/Software Reset	0x08	cmd	Invokes the software reset command.
Software Version	0x0C	get	Gets the current software version number.
Module Type	0x0E	get	Gets the module information, incl. module type with version number, chip version and laser type.
Module UID	0x0F	get	Gets the chip/module unique identification number.
Software Information / Identification	0x05	get	Gets the information about current software and device (e.g. version, device id, device family, ...)

5.5.2 Device Control Commands

Caption	Byte	Type	Comment
Measurement: Single Shot	0x10	cmd	Executes a single shot measurement.
Measurement: Start Auto	0x11	cmd	Starts the automatic, time-scheduled measurements with given frame rate.
Measurement: Stop	0x12	cmd	Stops the time-scheduled measurements (after the current frame finishes).
Calibration: Run	0x18	cmd	Executes a calibration sequence.
Re-Initialize Device	0x19	cmd	Invokes the device (re-)initialization command. Resets and reinitializes the API + ASIC with given config. (e.g. after unintended power cycle).

5.5.3 Measurement Data Commands

Two modes of streaming measurement data: on request or whenever a new data set is available. The actual data might depend on device configuration and calibration.

Caption	Byte	Type	Comment
Raw Measurement Data Set	0x30	get / auto/push	Gets a raw measurement data set containing the raw device readout samples.
Measurement Data Set (1D + 3D) - Debug	0x31	get / auto/push	Gets a measurement data set containing all the available data.
Measurement Data Set (1D + 3D)	0x32	get / auto/push	Gets a measurement data set containing the essential data.
3D Measurement Data Set - Debug	0x33	get / auto/push	Gets a 3D measurement data set containing all the available data per pixel.
3D Measurement Data Set	0x34	get / auto/push	Gets a 3D measurement data set containing the essential data per pixel.
1D Measurement Data Set - Debug	0x35	get / auto/push	Gets a 1D measurement data set containing all the available distance measurement data.
1D Measurement Data Set	0x36	get / auto/push	Gets a 1D measurement data set containing the essential distance measurement data.

5.5.4 Configuration Commands

Caption	Byte	Type	Comment
Data Output Mode	0x41	set / get	The measurement data output mode. (Hardware API only)
Measurement Mode	0x42	set / get	Gets or sets the measurement mode.
Frame Time (Frame Rate)	0x43	set / get	Gets or sets the measurement frame time (i.e. inverse frame rate).
Dual Frequency Mode	0x44	set / get	Gets or sets the dual frequency mode.
Smart Power Save Mode	0x45	set / get	Gets or sets the smart power saving feature enabled flag.
Shot Noise Monitor Mode	0x46	set / get	Gets or sets the shot noise monitor mode.
Dynamic Configuration Adaption	0x52	set / get	Gets or sets the full DCA feature configuration set.
Pixel Binning Algorithm	0x54	set / get	Gets or sets the pixel binning feature configuration.

Caption	Byte	Type	Comment
SPI Configuration	0x58	set / get	Gets or sets the SPI configuration (baud rate, slave instance).

5.5.5 Calibration Commands

Caption	Byte	Type	Comment
Global Range Offset	0x61	set / get	Gets or sets the global range offset calibration parameter.
Pixel Range Offsets	0x67	set / get	Gets or sets the crosstalk compensation vector table.
Pixel Range Offsets - Reset	0x68	cmd	Resets the crosstalk compensation vector table to factory calibrated default values.
Range Offsets Cal. Sequence - Sample Count	0x69	set / get	Gets or sets the crosstalk calibration sequence sample count.
Crosstalk Compensation - Vector Table	0x62	set / get	Gets or sets the crosstalk compensation vector table.
Crosstalk Compensation - Reset	0x63	cmd	Resets the crosstalk compensation vector table to factory calibrated default values.
Crosstalk Cal. Sequence - Sample Count	0x64	set / get	Gets or sets the crosstalk calibration sequence sample count.
Crosstalk Cal. Sequence - Max. Amplitude	0x65	set / get	Gets or sets the crosstalk calibration sequence maximum amplitude threshold.
Pixel-2-Pixel Crosstalk Compensation	0x66	set / get	Gets or sets the pixel-2-pixel crosstalk calibration parameter values.

5.6 Command Details

The following section contains the current command details as implemented in the *ExplorerApp*. The implementation can be found in the "Sources\explorer_app\api" folder of the "AFBR_S50_ExplorerApp_KL46z" project. See also Explorer App (API Demo).

5.6.1 Generic Commands

5.6.1.1 Invalid Command

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x00
Status	UINT8	1 / 0		Slave to Master only

5.6.1.2 Acknowledge

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x0A
Acknowledged Command	UINT8	1		

5.6.1.3 Not Acknowledge

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x0B
Not Acknowledged Command	UINT8	1		
Serial Status / Reason for NAK	UINT16	2		

5.6.1.4 Log Message

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x06
Time Stamp [sec]	UINT32	4	sec	
Time Stamp [µsec]	UINT16	2	ms/16	
Message	CHAR[] (= UINT8[])	n/a		

5.6.1.5 Test Message

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x04
Test message	UINT8[]	n/a		The returned message does also contain the CRC of the sent message.

5.6.1.6 MCU/Software Reset

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x08
Safety Code	UINT32	4		0xDEADC0DE

5.6.1.7 Software Version

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x0C
Version Software	UINT32	4		Major [31:24], Minor [23:16], Bugfix [15:0]
Software Build Number String	CHAR[14] (= UINT8[14])	n/a		(Added in v1.2.0) Software Build Number as String (14 Digits; e.g. "20200101123456")

Note

This command can be used to determine the current software version of the connected device. However, the command has changed as the build number was added to the command for v1.2.0 and newer versions. In order to determine the version, it is sufficient to encode only the version number. If the version is at least v1.2.0, the build number can also be extracted from the command data array.

5.6.1.8 Module Type

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x0E
Module Type/Version	UINT8	1		Module type number. See argus_module_version_t
Chip Type/Version	UINT8	1		Module type number. See argus_chip_version_t
Laser Type	UINT8	1		Module type number. See argus_laser_type_t

5.6.1.9 Module UID

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x0F
Module UID	UINT24	3		Unique Identification Number

5.6.1.10 Software Information / Identification

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x05
Version Software (Explorer App)	UINT32	4		Major [31:24], Minor [23:16], Bug-fix [15:0]
Version Argus API	UINT32	4		Major [31:24], Minor [23:16], Bug-fix [15:0]
Module Type/Version	UINT8	1		Module type number. See argus_module_version_t
Chip Type/Version	UINT8	1		Module type number. See argus_chip_version_t
Laser Type	UINT8	1		Module type number. See argus_laser_type_t
Module UID	UINT24	3		Unique Identification Number
Software ID String	CHAR[] (= UINT8[])	n/a		Software ID String incl. Build Number (Format: "AFBR-S50 Explorer App - [BuildNumber]"; w/ build number containing 14 digits, e.g. "20200101123456")

Note

This command can be used to determine the current software version of the connected device. However, the command has changed, e.g. the build number was added to the command for v1.2.0 and newer versions. In order to determine the version, it is sufficient to encode only the first version number. If the version is not current, the remaining command may differ. In case of different version, please refer to the corresponding API documentation to find the correct command formats.

5.6.2 Device Control Commands

5.6.2.1 Measurement: Trigger Single Shot

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x10

5.6.2.2 Measurement: Start Auto

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x11

5.6.2.3 Measurement: Stop

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x12

5.6.2.4 Run Calibration

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x18
Calibration Sequence ID	ENUM8	1	n/a	The selected calibration measurement sequence. See table below.
Optional Parameters	n/a	>=0	n/a	Optional parameters for the individual calibration sequences. See table below.

Value	Name	Parameters	Description
2	Crosstalk Calibration (Mode A)	n/a	Crosstalk calibration sequence for measurement mode A.
3	Crosstalk Calibration (Mode B)	n/a	Crosstalk calibration sequence for measurement mode B.
5	Pixel Range Offset Calibration (Mode A)	optional: Calibration Target Distance (Type: Q9.22; Unit: m)	Range offsets calibration sequence for measurement mode A.
6	Pixel Range Offset Calibration (Mode B)	optional: Calibration Target Distance (Type: Q9.22; Unit: m)	Range offsets calibration sequence for measurement mode B.

5.6.2.4.1 Calibration Sequence Enumerator

5.6.2.5 Device Reinitialize

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x19

5.6.3 Measurement Data Commands

5.6.3.1 Raw Measurement Data Set

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x30
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR < 0; STATUS > 0

Caption / Name	Type	Size	Unit	Comment
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.↩ e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	UQ10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	UQ12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
Enabled Pixel Mask	HEX32	4	n/a	The pixel enabled mask determines the pixels that are enabled and converted by the ADC muxing. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.
Enabled ADC Channel Mask	HEX32	4	n/a	The ADC channel enabled mask determines the misc. ADC channels that are enabled and converted by the ADC muxing. The mask is channel based (starting from channel 32), i.e. the n-th bit represents the n-th ADC channel. See the channel mapping for the corresponding channel purpose.
Phase Count	UINT8	1	#	Number of phase steps.
ADC Samples	UINT24[]	396	LSB	Raw sampling data from the pixel field. The 22 LSBs determine the raw 22-bit ADC readouts and the two MSBs determine the saturation flags. The values are ordered in increasing channel number where different phase steps are gathered (i.↩ e. $idx = 4n + p$). Disabled values (see Enabled Pixel Mask and Enabled ADC Channel Mask) are skipped.

5.6.3.2 Measurement Data Set (1D + 3D) - Debug

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x31
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; STATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	UQ10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	UQ12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
Enabled Pixel Mask	HEX32	4	n/a	The pixel enabled mask determines the pixels that are enabled and converted by the ADC muxing. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.
Enabled ADC Channel Mask	HEX32	4	n/a	The ADC channel enabled mask determines the misc. ADC channels that are enabled and converted by the ADC muxing. The mask is channel based (starting from channel 32), i.e. the n-th bit represents the n-th ADC channel. See the channel mapping for the corresponding channel purpuse.
Phase Count	UINT8	1	#	Number of phase steps.
ADC Samples	UINT24[]	396	LSB	Raw sampling data from the pixel field. The 22 LSBs determine the raw 22-bit ADC readouts and the two MSBs determine the saturation flags. The values are ordered in increasing channel number where different phase steps are gathered (i.e. $idx = 4n + p$). Disabled values (see Enabled Pixel Mask and Enabled ADC Channel Mask) are skipped.

Caption / Name	Type	Size	Unit	Comment
Status (x, y)	HEX8[,]	32	m	Status flags for each enabled pixel, see argus_px_status_t. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Range (x, y)	Q9.14[,]	96	m	Range values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Amplitude (x, y)	UQ12.4[,]	64		Amplitude values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Phase (x, y)	UQ1.15[,]	64	?	Phase values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Status (ref)	HEX8	1	m	Status flags (see argus_px_status_t) for the reference pixel (Skipped if disabled, see Enabled ADC Channel Mask).
Range (ref)	Q9.14	3	m	Range for the reference pixel (Skipped if disabled, see Enabled ADC Channel Mask).
Amplitude (ref)	UQ12.4	2		Amplitude for the reference pixel (Skipped if disabled, see Enabled ADC Channel Mask).
Phase (ref)	UQ1.15	2	?	Phase for the reference pixel (Skipped if disabled, see Enabled ADC Channel Mask).
1D Range (binned)	Q9.14	3	m	1D range as determined by the binning algorithm.
1D Amplitude (binned)	UQ12.4	2		1D amplitude as determined by the binning algorithm.
VDD	UQ12.4	2	LSB	Auxiliary measurement results for VDD
VDDL	UQ12.4	2	LSB	Auxiliary measurement results for VDDL
VSUB	UQ12.4	2	LSB	Auxiliary measurement results for VSUB
IAPD	UQ12.4	2	LSB	Auxiliary measurement results for IAPD
TEMP	Q11.4	2	℃	Auxiliary measurement results for TEMP
BGL	UQ12.4	2	LSB	Auxiliary estimation of the background light value.

5.6.3.3 Measurement Data Set (1D + 3D)

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x32
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; STATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	UQ10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	UQ12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
Enabled Pixel Mask	HEX32	4	n/a	The pixel enabled mask determines the pixels that are enabled and converted by the ADC muxing. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.
Status (x, y)	HEX8[,]	32	m	Status flags for each enabled pixel, see argus_px_status_t. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Range (x, y)	Q9.14[,]	96	m	Range values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Amplitude (x, y)	UQ12.4[,]	64		Amplitude values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
1D Range (binned)	Q9.14	3	m	1D range as determined by the binning algorithm.
1D Amplitude (binned)	UQ12.4	2		1D amplitude as determined by the binning algorithm.
VDD	UQ12.4	2	LSB	Auxiliary measurement results for VDD

Caption / Name	Type	Size	Unit	Comment
VDDL	UQ12.4	2	LSB	Auxiliary measurement results for VDDL
VSUB	UQ12.4	2	LSB	Auxiliary measurement results for VSUB
IAPD	UQ12.4	2	LSB	Auxiliary measurement results for IAPD
TEMP	Q11.4	2	℃	Auxiliary measurement results for TEMP
BGL	UQ12.4	2	LSB	Auxiliary estimation of the background light value.

5.6.3.4 3D Measurement Data Set - Debug

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x33
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; STATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.↩ e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	UQ10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	UQ12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
Enabled Pixel Mask	HEX32	4	n/a	The pixel enabled mask determines the pixels that are enabled and converted by the ADC muxing. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.
Status (x, y)	HEX8[,]	32	m	Status flags for each enabled pixel, see argus_px_status_t. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.

Caption / Name	Type	Size	Unit	Comment
Range (x, y)	Q9.14[,]	96	m	Range values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Amplitude (x, y)	UQ12.4[,]	64		Amplitude values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Phase (x, y)	UQ1.15[,]	64	?	Phase values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.

5.6.3.5 3D Measurement Data Set

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x34
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; STATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	UQ10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	UQ12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
Enabled Pixel Mask	HEX32	4	n/a	The pixel enabled mask determines the pixels that are enabled and converted by the ADC muxing. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.

Caption / Name	Type	Size	Unit	Comment
Status (x, y)	HEX8[,]	32	m	Status flags for each enabled pixel, see argus_px_status_t. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Range (x, y)	Q9.14[,]	96	m	Range values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.
Amplitude (x, y)	UQ12.4[,]	64		Amplitude values for each enabled pixel. The values are ordered in increasing x and y indices (i.e. $n = 4x + y$). Disabled values (see Enabled Pixel Mask) are skipped.

5.6.3.6 1D Measurement Data Set - Debug

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x35
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; ST↩ ATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
Digital Integration Depth	UINT16	2	#	The digital integration depth determines the digital averaging depth of a single measurement frame. This is the averaging sample count i.e. the number of repeated analog measurements for each phase step.
Analog Integration Depth	U↩ Q10.6	2	#	The analog integration depth determines the number of 128-bit laser patterns that are sent for a single correlation cycle. The value can be either a multiple of a full pattern or a fraction of a single pattern (i.e. a multiple of single pulses/periods).
Optical Power	U↩ Q12.4	2	mA	The optical output power is determined by the laser current.
Pixel Gain	UINT8	1		The pixel gain determines the sensitivity of the ToF cells.
1D Pixel Count	UINT8	1	#	The number of pixels considered for the 1D range value.
Saturated Pixel Count	UINT8	1	#	The number of pixels with saturation flags set.
1D Range (binned)	Q9.14	3	m	1D range as determined by the binning algorithm.
1D Amplitude (binned)	U↩ Q12.4	2	LSB	1D amplitude as determined by the binning algorithm.
1D Phase (binned)	UQ1.15	2	?	1D phase as determined by the binning algorithm.

5.6.3.7 1D Measurement Data Set

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x36
Status	HEX16	2	n/a	Provides information about the measurement status. OK = 0; ERROR $<$ 0; ST↩ ATUS $>$ 0
Timestamp	UINT48	6	sec;µsec/16	Contains the measurement start time.
Measurement Frame State Flags	HEX16	2	n/a	The state of the current measurement frame. See argus_state_t for details.
1D Range (binned)	Q9.14	3	m	1D range as determined by the binning algorithm.
1D Amplitude (binned)	U↩ Q12.4	2		1D amplitude as determined by the binning algorithm.

5.6.4 Configuration Commands

5.6.4.1 Data Output Mode

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x41
Measurement Data Output Mode	ENUM8	1	n/a	The measurement data output mode. See the table below for details.

Value	Name	Description
1	Streaming Raw Data	When in 'Raw Data Streaming Mode', the software is streaming the raw measurement data only, i.e. the raw correlation samples per pixel.
2	Streaming Debug Data	When in 'Debug Data Streaming Mode', the software is streaming all available measurement data from the 1D and 3D measurements, i.e. raw correlation sampling data as well as the range, phase and amplitude values per pixel (3D) and from the pixel binning algorithm (1D). Additional information about the measurement frame is also provided. If enabled, also the reference pixel results are available.
3	Streaming Full Data	When in 'Full Data Streaming Mode', the software is streaming all essential measurement data from the 1D and 3D measurements, i.e. range and amplitude values per pixel (3D) as well as from the pixel binning algorithm (1D). Additional information about the measurement frame is also provided.
4	Streaming 3D Debug Data	When in '3D Debug Data Streaming Mode', the software is streaming all available measurement data from the 3D measurements, i.e. the range, phase and amplitude values per pixel (3D). Additional information about the measurement frame is also provided.
5	Streaming 3D Data	When in '3D Data Streaming Mode', the software is streaming all essential measurement data from the 3D measurements, i.e. range and amplitude values per pixel (3D). Additional information about the measurement frame is also provided.
6	Streaming 1D Debug Data	When in '1D Debug Data Streaming Mode', the software is streaming all available measurement data from the 1D measurements, i.e. the range, phase and amplitude values from the pixel binning algorithm (1D). Additional information about the measurement frame is also provided.

Value	Name	Description
7	Streaming 1D Data	When in '1D Data Streaming Mode', the software is streaming all essential measurement data from the 1D measurements, i.e. range and amplitude values from the pixel binning algorithm (1D).

5.6.4.1.1 Measurement Data Output Mode Enumerator

5.6.4.2 Measurement Mode

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x42
Measurement Mode	ENUM8	1	n/a	Gets or sets the measurement mode. See argus_mode_t for details.

5.6.4.3 Frame Time

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x43
Frame Time (inverse Frame Rate)	UINT32	4	µsec	The measurement frame time (1 / "frame rate") in usec. The interval to trigger measurement frames.

5.6.4.4 Dual Frequency Mode

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x44
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Dual Frequency Mode	ENUM8	1	n/a	Selects the Dual-Frequency Mode (DFM). See See argus_dfm_mode_t or the table below for details.

Value	Name	Description
0	1x Mode	Single Frequency Measurement Mode (w/ 1x Unambiguous Range). The DFM is disabled.
1	4x Mode	4X Dual Frequency Measurement Mode (w/ 4x Unambiguous Range).
2	8x Mode	8X Dual Frequency Measurement Mode (w/ 8x Unambiguous Range).

5.6.4.4.1 Dual Frequency Mode Enumerator

5.6.4.5 Smart Power Save Mode

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x45
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Smart Power Save Enabled	BOOL8	1	n/a	Enables the Smart Power Save Feature.

5.6.4.6 Shot Noise Monitor Mode

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x46
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Shot Noise Monitor Mode	ENUM8	1	n/a	Selects the Shot-Noise-Monitor (SNM) mode. See argus_snm_mode_t or the table below for details.

Value	Name	Description
0	Static (Indoor)	Static Shot Noise Monitoring Mode, optimized for indoor applications.
1	Static (Outdoor)	Static Shot Noise Monitoring Mode, optimized for outdoor applications.
2	Dynamic	Dynamic Shot Noise Monitoring Mode.

5.6.4.6.1 Shot Noise Monitor Mode Enumerator

5.6.4.7 Dynamic Configuration Adaption

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x52
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Enabled Flags	HEX8	1	n/a	DCA features enable flags. Bit 0: DCA feature enable, Bit 1: use average amplitude
Saturated Pixel Threshold for Linear Response	UINT8	1	#	The maximum allowed overflow pixels before the DCA will invoke an linear decrease of integration energy.
Saturated Pixel Threshold for Exponential Response	UINT8	1	#	The maximum allowed overflow pixels before the DCA will invoke an exponential decrease of integration energy.
Saturated Pixel Threshold for Reset Response	UINT8	1	#	The maximum allowed overflow pixels before the DCA will invoke an integration energy reset. Use 32 to disable.
Target Amplitude	U Q12.4	2	LSB	The target amplitude. The DCA will try to linearity approach the target value.
Low Amplitude Threshold	U Q12.4	2	LSB	The minimum allowed amplitude before the integration energy will be increased.
High Amplitude Threshold	U Q12.4	2	LSB	The maximum allowed amplitude before the integration energy will be decreased.
Nominal Integration Depth	U Q10.6	2	#	The nominal analog integration depth. This determines the start value or the static value depending on the Enabled setting.
Min. Integration Depth	U Q10.6	2	#	The minimum analog integration depth, i.e. the minimum pattern count per sample.
Max. Integration Depth	U Q10.6	2	#	The maximum analog integration depth, i.e. the maximum pattern count per sample.

Caption / Name	Type	Size	Unit	Comment
Nominal Optical Power	U↩ Q12.4	2	mA	The nominal optical output power.This determines the start value or the static value depending on the Enabled setting.
Min. Optical Power	U↩ Q12.4	2	mA	The minimum optical output power, i.e. the minimum OMA value.
Nominal Pixel Gain	UINT8	1	n/a	The nominal pixel gain value for the default DCA gain stage. This determines the start value or the static value depending on the Enabled setting.
Low Pixel Gain	UINT8	1	n/a	The low pixel gain value for the DCA low gain stage.
High Pixel Gain	UINT8	1	n/a	The high pixel gain value for the DCA high gain stage.
Power Saving Ratio	UQ0.8	1	n/a	Determines the percentage of the full available frame time that is not exploited for digital integration. Thus the device is idle within the specified portion of the frame time and does consume less energy.

5.6.4.8 Pixel Binning

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x54
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Enabled Flags	HEX8	1	n/a	Enables/disables the pixel binning (features). See argus_pba_flags_t for details.
Averaging Mode	HEX8	1	n/a	Selects the averaging mode for the PBA. See argus_pba_averaging_mode_t for details.
Pre-filter Mask	HEX32	4	n/a	The pixel pre-filter mask determines the pixels that are dismissed by the PBA. The mask is channel based, i.e. the n-th bit represents the n-th ADC channel. See the pixel mapping for the corresponding x-y-indices.
Absolute Amplitude Threshold	U↩ Q12.4	2	LSB	Absolute Amplitude Threshold in LSB. Lower values are dismissed.
Relative Amplitude Threshold	UQ0.8	1	%	Relative Amplitude Threshold in % of max value. Lower values are dismissed.
Absolute Minimum Distance Scope	UQ1.15	2	m	Absolute Minimum Distance Scope Filter Threshold Value in LSB. Distances with larger values (compared to current minimum distance) are dismissed.
Relative Minimum Distance Scope	UQ0.8	1	%	Relative Minimum Distance Scope Filter Threshold Value in % of the current minimum distance. Distances with larger values (compared to current minimum distance) are dismissed

5.6.4.9 SPI Configuration

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x58
SPI Slave	ENUM8	1	0	SPI Slave Enumeration: 0: intern, 1: extern,
SPI Baud Rate	UINT32	4	0	Baud Rate in BaudPerSeconds.

5.6.5 Calibration Commands

5.6.5.1 Global Range Offset

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x61
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Range Offset	Q9.22	4	m	The global range offset in meters.

5.6.5.2 Pixel Range Offsets

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x67
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Offset Values (x, y)	Q0.15	64	m	The pixel offsets table. Ordering: X Index -> Y Index.

Note on indices:

- x: The pixel x-index (0-7)
- y: The pixel y-index (0-3)

5.6.5.3 Pixel Range Offsets - Reset Table

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x68
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.

5.6.5.4 Range Offsets Calibration Sequence - Sample Count

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x69
Sample Count	UINT16	2		The number of sample per offset calibration measurement.

5.6.5.5 Crosstalk Compensation - Vector Table

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x62
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.
Crosstalk Vector (f, x, y, a)	Q9.22	256	LSB	The crosstalk vector table. Ordering: Frequency Index (A/B) -> X Index -> Y Index -> Sin/Cos Index.

Note on indices:

- f: The frequency index for A- and B-frames respectively (A = 0; B = 1)
- x: The pixel x-index (0-7)
- y: The pixel y-index (0-3)
- a: The sin/cos index (sin = 0; cos = 1)

5.6.5.6 Crosstalk Compensation - Reset Vector Table

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x63
Measurement Mode Selector	ENUM8	1	n/a	The addressed measurement mode. See argus_mode_t for details.

5.6.5.7 Crosstalk Calibration Sequence - Sample Count

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x64
Sample Count	UINT16	2		The number of sample per crosstalk calibration measurement.

5.6.5.8 Crosstalk Calibration Sequence - Maximum Amplitude Threshold

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x65
Maximum Amplitude Threshold	U↩ Q12.4	2	LSB	The maximum amplitude threshold for crosstalk calibration sequence.

5.6.5.9 Pixel-2-Pixel Crosstalk Compensation Parameters

Caption / Name	Type	Size	Unit	Comment
Command	UINT8	1		0x66
Measurement Mode Selector	ENUM8	1		The addressed measurement mode.
Enabled	BOOL8	1		Enables/disables the pixel-2-pixel crosstalk compensation algorithm.
Kc Factor	UQ0.8	1		The Kc factor that determines the amount of the total signal on all pixels influences the individual signal of each pixel.

Caption / Name	Type	Size	Unit	Comment
Relative Threshold	UQ0.8	1		The relative threshold determines when the compensation is active for each individual pixel.
Absolute Threshold	U↩ Q12.4	2		The absolute threshold determines the minimum total crosstalk amplitude that is required for the compensation to become active.

Chapter 6

Module Index

6.1 Modules

Here is a list of all modules:

Chapter 7

Data Structure Index

7.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all documented files with brief descriptions:

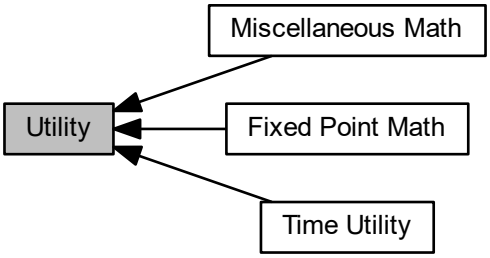
Chapter 9

Module Documentation

9.1 Utility

Utility library containing useful functions.

Collaboration diagram for Utility:



Modules

- Miscellaneous Math

  Miscellaneous math utility functions utilized within the API.
- Fixed Point Math

  A basic math library for fixed point number in the Qx.y fomat.
- Time Utility

  Timer utilities for time measurement duties.

9.1.1 Detailed Description

Utility library containing useful functions.

Contains all utility code, e.g. fixed-point math, that belongs to the AFBR-S50 core library algorithms.

The methods and definitions will be helpful for user when implementing their own usage of the AFBR-S50 API.

9.2 Miscellaneous Math

Miscellaneous math utility functions utilized within the API.
Collaboration diagram for Miscellaneous Math:

Macros

- #define INT_SQRT 0
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

  Calculates the maximum of two values.
- #define MIN(a, b) ((a) < (b) ? (a) : (b))

  Calculates the minimum of two values.

9.2.1 Detailed Description

Miscellaneous math utility functions utilized within the API.
Modules overview:

- Integer Math: Mathematical functionality on integer values.
- Long integer multiplication (32bit x 32bit): double word integer multiplication algorithms.

9.2.2 Macro Definition Documentation

9.2.2.1 INT_SQRT

```
#define INT_SQRT 0
```

Enables the integer square root function.

9.2.2.2 MAX

```
#define MAX(
              a,
              b ) ((a) > (b) ?  (a) :  (b))
```

Calculates the maximum of two values.

Parameters

a	Input parameter.
b	Input parameter.

Returns

The maximum value of the input parameters.

9.2.2.3 MIN

```
#define MIN(
              a,
              b ) ((a) < (b) ?  (a) :  (b))
```

Calculates the minimum of two values.

Parameters

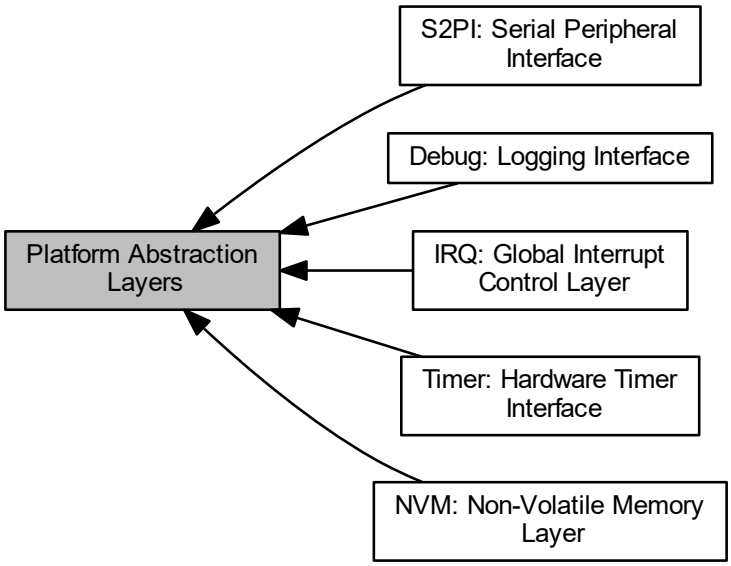
a	Input parameter.
b	Input parameter.

Returns

The minimum value of the input parameters.

9.3 Platform Abstraction Layers

Collection of all interfaces for the Platform Abstraction Layers (HAL).
Collaboration diagram for Platform Abstraction Layers:



Modules

- IRQ: Global Interrupt Control Layer

  Global Interrupt Control Layer.
- NVM: Non-Volatile Memory Layer

  Non-Volatile Memory Layer.
- Debug: Logging Interface

  Logging interface for the AFBR-S50 API.
- S2PI: Serial Peripheral Interface

  S2PI: SPI incl. GPIO Hardware Layer Module.
- Timer: Hardware Timer Interface

  Timer implementations for lifetime counting as well as periodic callback.

9.3.1 Detailed Description

Collection of all interfaces for the Platform Abstraction Layers (HAL).
Contains all platform dependent code required by the API and core library.

9.4 Configuration

Device configuration parameter definitions and API functions.
Collaboration diagram for Configuration:

Functions

- status_t Argus_SetConfigurationMeasurementMode (argus_hnd_t *hnd, argus_mode_t value)

  Sets the measurement mode to a specified device.
- status_t Argus_GetConfigurationMeasurementMode (argus_hnd_t *hnd, argus_mode_t *value)

  Gets the measurement mode from a specified device.
- status_t Argus_SetConfigurationFrameTime (argus_hnd_t *hnd, uint32_t value)

  Sets the frame time to a specified device.
- status_t Argus_GetConfigurationFrameTime (argus_hnd_t *hnd, uint32_t *value)

  Gets the frame time from a specified device.
- status_t Argus_SetConfigurationSmartPowerSaveEnabled (argus_hnd_t *hnd, argus_mode_t mode, bool value)

  Sets the smart power save enabled flag to a specified device.
- status_t Argus_GetConfigurationSmartPowerSaveEnabled (argus_hnd_t *hnd, argus_mode_t mode, bool *value)

  Gets the smart power save enabled flag from a specified device.
- status_t Argus_SetConfigurationDFMMode (argus_hnd_t *hnd, argus_mode_t mode, argus_dfm_mode_t value)

  Sets the Dual Frequency Mode (DFM) to a specified device.
- status_t Argus_GetConfigurationDFMMode (argus_hnd_t *hnd, argus_mode_t mode, argus_dfm_mode_t *value)

  Gets the Dual Frequency Mode (DFM) from a specified device.
- status_t Argus_SetConfigurationShotNoiseMonitorMode (argus_hnd_t *hnd, argus_mode_t mode, argus_snm_mode_t value)

  Sets the Shot Noise Monitor (SNM) mode to a specified device.
- status_t Argus_GetConfigurationShotNoiseMonitorMode (argus_hnd_t *hnd, argus_mode_t mode, argus_snm_mode_t *value)

  Gets the Shot Noise Montor (SNM) mode from a specified device.
- status_t Argus_SetConfigurationDynamicAdaption (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_dca_t const *value)

  Sets the full DCA module configuration to a specified device.
- status_t Argus_GetConfigurationDynamicAdaption (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_dca_t *value)

  Gets the # from a specified device.
- status_t Argus_SetConfigurationPixelBinning (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_pba_t const *value)

  Sets the pixel binning configuration parameters to a specified device.
- status_t Argus_GetConfigurationPixelBinning (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_pba_t *value)

Gets the pixel binning configuration parameters from a specified device.

- status_t Argus_GetConfigurationUnambiguousRange (argus_hnd_t *hnd, uint32_t *range_mm)

  Gets the current unambiguous range in mm.

9.4.1 Detailed Description

Device configuration parameter definitions and API functions.
This module takes care of keeping the device configuration up to date. Therefore, the configuration is managed in a local data structure. Requested changes to the device configuration are validated and endorsed within the module.
Configuration API

9.4.2 Function Documentation

9.4.2.1 Argus_GetConfigurationDFMMode()

```
status_t Argus_GetConfigurationDFMMode (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_dfm_mode_t * value )
```

Gets the Dual Frequency Mode (DFM) from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current DFM mode value.

Returns

Returns the status (STATUS_OK on success).

9.4.2.2 Argus_GetConfigurationDynamicAdaption()

```
status_t Argus_GetConfigurationDynamicAdaption (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cfg_dca_t * value )
```

Gets the # from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current DCA configuration set value.

Returns

Returns the status (STATUS_OK on success).

9.4.2.3 Argus_GetConfigurationFrameTime()

```
status_t Argus_GetConfigurationFrameTime (
```

```
            argus_hnd_t * hnd,
            uint32_t * value )
```

Gets the frame time from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current frame time in microseconds.

Returns

Returns the status (STATUS_OK on success).

9.4.2.4 Argus_GetConfigurationMeasurementMode()

```
status_t Argus_GetConfigurationMeasurementMode (
            argus_hnd_t * hnd,
            argus_mode_t * value )
```

Gets the measurement mode from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.4.2.5 Argus_GetConfigurationPixelBinning()

```
status_t Argus_GetConfigurationPixelBinning (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cfg_pba_t * value )
```

Gets the pixel binning configuration parameters from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current pixel binning configuration parameters.

Returns

Returns the status (STATUS_OK on success).

9.4.2.6 Argus_GetConfigurationShotNoiseMonitorMode()

```
status_t Argus_GetConfigurationShotNoiseMonitorMode (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_snm_mode_t * value )
```

Gets the Shot Noise Montor (SNM) mode from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current SNM mode value.

Returns

Returns the status (STATUS_OK on success).

9.4.2.7 Argus_GetConfigurationSmartPowerSaveEnabled()

```
status_t Argus_GetConfigurationSmartPowerSaveEnabled (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            bool * value )
```

Gets the smart power save enabled flag from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current smart power save enabled flag.

Returns

Returns the status (STATUS_OK on success).

9.4.2.8 Argus_GetConfigurationUnambiguousRange()

```
status_t Argus_GetConfigurationUnambiguousRange (
            argus_hnd_t * hnd,
            uint32_t * range_mm )
```

Gets the current unambiguous range in mm.

Parameters

hnd	The API handle; contains all internal states and data.
range_mm	The returned range in mm.

Returns

Returns the status (STATUS_OK on success).

9.4.2.9 Argus_SetConfigurationDFMMode()

```
status_t Argus_SetConfigurationDFMMode (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_dfm_mode_t value )
```

Sets the Dual Frequency Mode (DFM) to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new DFM mode value.

Returns

Returns the status (STATUS_OK on success).

9.4.2.10 Argus_SetConfigurationDynamicAdaption()

```
status_t Argus_SetConfigurationDynamicAdaption (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cfg_dca_t const * value )
```

Sets the full DCA module configuration to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new DCA configuration set.

Returns

Returns the status (STATUS_OK on success).

9.4.2.11 Argus_SetConfigurationFrameTime()

```
status_t Argus_SetConfigurationFrameTime (
            argus_hnd_t * hnd,
            uint32_t value )
```

Sets the frame time to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
value	The measurement frame time in microseconds.

Returns

Returns the status (STATUS_OK on success).

Examples

01_simple_example.c, and 02_advanced_example.c.

9.4.2.12 Argus_SetConfigurationMeasurementMode()

```
status_t Argus_SetConfigurationMeasurementMode (
```

```
            argus_hnd_t * hnd,
            argus_mode_t value )
```

Sets the measurement mode to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
value	The new measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.4.2.13 Argus_SetConfigurationPixelBinning()

```
status_t Argus_SetConfigurationPixelBinning (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cfg_pba_t const * value )
```

Sets the pixel binning configuration parameters to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new pixel binning configuration parameters.

Returns

Returns the status (STATUS_OK on success).

9.4.2.14 Argus_SetConfigurationShotNoiseMonitorMode()

```
status_t Argus_SetConfigurationShotNoiseMonitorMode (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_snm_mode_t value )
```

Sets the Shot Noise Monitor (SNM) mode to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new SNM mode value.

Returns

Returns the status (STATUS_OK on success).

9.4.2.15 Argus_SetConfigurationSmartPowerSaveEnabled()

```
status_t Argus_SetConfigurationSmartPowerSaveEnabled (
            argus_hnd_t * hnd,
```

```
            argus_mode_t mode,
            bool value )
```

Sets the smart power save enabled flag to a specified device.

Parameters

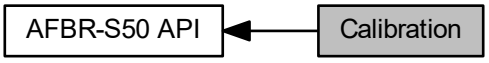
hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new smart power save enabled flag.

Returns

Returns the status (STATUS_OK on success).

9.5 Calibration

Device calibration parameter definitions and API functions.
Collaboration diagram for Calibration:



Data Structures

- struct xtalk_t

  Pixel Crosstalk Compensation Vector.
- struct argus_cal_p2pxtalk_t

  Pixel-To-Pixel Crosstalk Compensation Parameters.

Functions

- status_t Argus_SetCalibrationGlobalRangeOffset (argus_hnd_t *hnd, argus_mode_t mode, q9_22_t value)

  Sets the global range offset value to a specified device.
- status_t Argus_GetCalibrationGlobalRangeOffset (argus_hnd_t *hnd, argus_mode_t mode, q9_22_t *value)

  Gets the global range offset value from a specified device.
- status_t Argus_SetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Sets the relative pixel offset table to a specified device.
- status_t Argus_GetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the relative pixel offset table from a specified device.
- status_t Argus_GetCalibrationTotalPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the relative pixel offset table from a specified device.
- status_t Argus_ResetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode)

  Resets the relative pixel offset values for the specified device to the factory calibrated default values.
- void Argus_GetExternalPixelRangeOffsets_Callback (q0_15_t offsets[ARGUS_PIXELS_X][ARGUS_PIXELS_Y], argus_mode_t mode)

  A callback that returns the external pixel range offsets.
- status_t Argus_SetCalibrationRangeOffsetSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the range offset calibration sequence.
- status_t Argus_GetCalibrationRangeOffsetSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the range offset calibration sequence.
- status_t Argus_SetCalibrationCrosstalkPixel2Pixel (argus_hnd_t *hnd, argus_mode_t mode, argus_cal_p2pxtalk_t const *value)

  Sets the pixel-to-pixel crosstalk compensation parameters to a specified device.
- status_t Argus_GetCalibrationCrosstalkPixel2Pixel (argus_hnd_t *hnd, argus_mode_t mode, argus_cal_p2pxtalk_t *value)

  Gets the pixel-to-pixel crosstalk compensation parameters from a specified device.
- status_t Argus_SetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

Sets the custom crosstalk vector table to a specified device.

- status_t Argus_GetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the custom crosstalk vector table from a specified device.
- status_t Argus_GetCalibrationTotalCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the factory calibrated default crosstalk vector table for the specified device.
- status_t Argus_ResetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode)

  Resets the crosstalk vector table for the specified device to the factory calibrated default values.
- status_t Argus_SetCalibrationCrosstalkSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the crosstalk calibration sequence.
- status_t Argus_GetCalibrationCrosstalkSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the crosstalk calibration sequence.
- status_t Argus_SetCalibrationCrosstalkSequenceAmplitudeThreshold (argus_hnd_t *hnd, uq12_4_t value)

  Sets the max. amplitude threshold for the crosstalk calibration sequence.
- status_t Argus_GetCalibrationCrosstalkSequenceAmplitudeThreshold (argus_hnd_t *hnd, uq12_4_t *value)

  Gets the max. amplitude threshold for the crosstalk calibration sequence.
- status_t Argus_SetCalibrationVsubSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the substrate voltage calibration sequence.
- status_t Argus_GetCalibrationVsubSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the substrate voltage calibration sequence.
- void Argus_GetExternalCrosstalkVectorTable_Callback (xtalk_t xtalk[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][AR argus_mode_t mode)

  A callback that returns the external crosstalk vector table.

9.5.1 Detailed Description

Device calibration parameter definitions and API functions.
The calibration concept of the Time-of-Flight measurement device is made up of three parts:

- Factory calibration data which is provided in the devices EEPROM. This data is read and applied upon device initialization.
- Online calibration is executed along with the actual distance measurements. Therefore, ambient information, e.g. temperature or voltage levels, are gathered during measurements. The information is then utilized to adjust the device configuration and evaluation algorithms accordingly.
- User calibration parameters are applied on top of this. These parameters are provided in order to adopt the device to the user application, e.g. cover glass calibration. These parameters must be set (if required) by the given API.

Calibration API

9.5.2 Function Documentation

9.5.2.1 Argus_GetCalibrationCrosstalkPixel2Pixel()

```
status_t Argus_GetCalibrationCrosstalkPixel2Pixel (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cal_p2pxtalk_t * value )
```

Gets the pixel-to-pixel crosstalk compensation parameters from a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current pixel-to-pixel crosstalk compensation parameters.

Returns

Returns the status (STATUS_OK on success).

9.5.2.2 Argus_GetCalibrationCrosstalkSequenceAmplitudeThreshold()

```
status_t Argus_GetCalibrationCrosstalkSequenceAmplitudeThreshold (
            argus_hnd_t * hnd,
            uq12_4_t * value )
```

Gets the max. amplitude threshold for the crosstalk calibration sequence.

The maximum amplitude threshold defines a maximum crosstalk vector amplitude before causing an error message. If the crosstalk is too high, there is usually an issue with the measurement setup, i.e. there is still a measurement signal detected.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current max. amplitude threshold value in UQ12.4 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.3 Argus_GetCalibrationCrosstalkSequenceSampleCount()

```
status_t Argus_GetCalibrationCrosstalkSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t * value )
```

Gets the sample count for the crosstalk calibration sequence.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current crosstalk calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.5.2.4 Argus_GetCalibrationCrosstalkVectorTable()

```
status_t Argus_GetCalibrationCrosstalkVectorTable (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Gets the custom crosstalk vector table from a specified device.

The crosstalk vectors are subtracted from the raw sampling data in the data evaluation phase.
The crosstalk vector table is a three dimensional array of type xtalk_t.
The dimensions are:

- size(0) = ARGUS_DFM_FRAME_COUNT (Dual-frequency mode A- or B-frame)
- size(1) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(2) = ARGUS_PIXELS_Y (Pixel count in y-direction)

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current crosstalk vector table.

Returns

Returns the status (STATUS_OK on success).

9.5.2.5 Argus_GetCalibrationGlobalRangeOffset()

```
status_t Argus_GetCalibrationGlobalRangeOffset (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            q9_22_t * value )
```

Gets the global range offset value from a specified device.
The global range offset is subtracted from the raw range values.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current global range offset in meter and Q9.22 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.6 Argus_GetCalibrationPixelRangeOffsets()

```
status_t Argus_GetCalibrationPixelRangeOffsets (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Gets the relative pixel offset table from a specified device.
The relative pixel offset values are subtracted from the raw range values for each individual pixel. Note that a global range offset is applied additionally. The relative pixel offset values are meant to be with respect to the average range of all pixels, i.e. the average of all relative offsets should be 0!
The crosstalk vector table is a two dimensional array of type q0_15_t.
The dimensions are:

- size(0) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(1) = ARGUS_PIXELS_Y (Pixel count in y-direction)

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current relative range offset in meter and Q0.15 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.7 Argus_GetCalibrationRangeOffsetSequenceSampleCount()

```
status_t Argus_GetCalibrationRangeOffsetSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t * value )
```

Gets the sample count for the range offset calibration sequence.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current range offset calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.5.2.8 Argus_GetCalibrationTotalCrosstalkVectorTable()

```
status_t Argus_GetCalibrationTotalCrosstalkVectorTable (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Gets the factory calibrated default crosstalk vector table for the specified device.
The crosstalk vectors are subtracted from the raw sampling data in the data evaluation phase.
The crosstalk vector table is a three dimensional array of type xtalk_t.
The dimensions are:

- size(0) = ARGUS_DFM_FRAME_COUNT (Dual-frequency mode A- or B-frame)
- size(1) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(2) = ARGUS_PIXELS_Y (Pixel count in y-direction)

The total vector table consists of the custom crosstalk vector table (set via Argus_SetCalibrationCrosstalkVectorTable) and an internal, factory calibrated device specific vector table. This is informational only!

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current total crosstalk vector table.

Returns

Returns the status (STATUS_OK on success).

9.5.2.9 Argus_GetCalibrationTotalPixelRangeOffsets()

```
status_t Argus_GetCalibrationTotalPixelRangeOffsets (
            argus_hnd_t * hnd,
```

```
            argus_mode_t mode,
            q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Gets the relative pixel offset table from a specified device.

The relative pixel offset values are subtracted from the raw range values for each individual pixel. Note that a global range offset is applied additionally. The relative pixel offset values are meant to be with respect to the average range of all pixels, i.e. the average of all relative offsets should be 0!
The crosstalk vector table is a two dimensional array of type q0_15_t.
The dimensions are:

- size(0) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(1) = ARGUS_PIXELS_Y (Pixel count in y-direction)

The total offset table consists of the custom pixel offset values (set via Argus_SetCalibrationPixelRangeOffsets) and the internal, factory calibrated device specific offset values. This is informational only!

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The current total relative range offset in meter and Q0.15 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.10 Argus_GetCalibrationVsubSequenceSampleCount()

```
status_t Argus_GetCalibrationVsubSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t * value )
```

Gets the sample count for the substrate voltage calibration sequence.

Parameters

hnd	The API handle; contains all internal states and data.
value	The current substrate voltage calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.5.2.11 Argus_GetExternalCrosstalkVectorTable_Callback()

```
void Argus_GetExternalCrosstalkVectorTable_Callback (
            xtalk_t xtalk[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y],
            argus_mode_t mode )
```

A callback that returns the external crosstalk vector table.

The function needs to be implemented by the host application in order to set the external crosstalk vector table upon system initialization. If not defined in user code, the default implementation will return an all zero vector table, assuming there is no (additional) external crosstalk.
If defined in user code, the function must fill all vector values in the provided

xtalk parameter with external crosstalk values.

The values can be obtained by the calibration routine.

Example usage:

```
status_t Argus_GetExternalCrosstalkVectorTable_Callback(xtalk_t
       xtalk[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y],
                                                        argus_mode_t mode)
{
    (void) mode; // Ignore mode; use same values for all modes.
    memset(&xtalk, 0, sizeof(xtalk));
    // Set crosstalk vectors in Q11.4 format.
    // Note on dual-frequency frame index: 0 = A-Frame; 1 = B-Frame
    xtalk[0][0][0].dS = -9;      xtalk[0][0][0].dC = -11;
    xtalk[0][0][1].dS = -13;     xtalk[0][0][1].dC = -16;
    xtalk[0][0][2].dS = 6;       xtalk[0][0][2].dC = -18;
    // etc.
}
```

Parameters

xtalk	The crosstalk vector array; to be filled with data.
mode	Determines the current measurement mode; can be ignored if only a single measurement mode is utilized.

9.5.2.12 Argus_GetExternalPixelRangeOffsets_Callback()

```
void Argus_GetExternalPixelRangeOffsets_Callback (
             q0_15_t offsets[ARGUS_PIXELS_X][ARGUS_PIXELS_Y],
             argus_mode_t mode )
```

A callback that returns the external pixel range offsets.

The function needs to be implemented by the host application in order to set the external pixel range offsets values upon system initialization. If not defined in user code, the default implementation will return an all zero offset table, assuming there is no (additional) external pixel range offset values.
If defined in user code, the function must fill all offset values in the provided

offsets parameter with external range offset

values. The values can be obtained by the calibration routine.

Example usage:

```
status_t Argus_GetExternalPixelRangeOffsets_Callback(q0_15_t offsets[ARGUS_PIXELS_X][ARGUS_PIXELS_Y],
                                                     argus_mode_t mode)
{
    (void) mode; // Ignore mode; use same values for all modes.
    memset(offsets, 0, sizeof(q0_15_t) * ARGUS_PIXELS);
    // Set offset values in meter and Q0.15 format.
    offsets[0][0].dS = -16384;      offsets[0][0].dC = -32768;
    offsets[0][1].dS = -32768;      offsets[0][1].dC = 0;
    offsets[0][2].dS = 16384;       offsets[0][2].dC = -16384;
    // etc.
}
```

Parameters

offsets	The pixel range offsets in meter and Q0.15 format; to be filled with data.
mode	Determines the current measurement mode; can be ignored if only a single measurement mode is utilized.

9.5.2.13 Argus_ResetCalibrationCrosstalkVectorTable()

```
status_t Argus_ResetCalibrationCrosstalkVectorTable (
             argus_hnd_t * hnd,
             argus_mode_t mode )
```

Resets the crosstalk vector table for the specified device to the factory calibrated default values.
The crosstalk vectors are subtracted from the raw sampling data in the data evaluation phase.

- The factory defaults are device specific calibrated values.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.5.2.14 Argus_ResetCalibrationPixelRangeOffsets()

```
status_t Argus_ResetCalibrationPixelRangeOffsets (
            argus_hnd_t * hnd,
            argus_mode_t mode )
```

Resets the relative pixel offset values for the specified device to the factory calibrated default values.
The relative pixel offset values are subtracted from the raw range values for each individual pixel. Note that a global range offset is applied additionally.
The factory defaults are device specific values.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.5.2.15 Argus_SetCalibrationCrosstalkPixel2Pixel()

```
status_t Argus_SetCalibrationCrosstalkPixel2Pixel (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            argus_cal_p2pxtalk_t const * value )
```

Sets the pixel-to-pixel crosstalk compensation parameters to a specified device.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new pixel-to-pixel crosstalk compensation parameters.

Returns

Returns the status (STATUS_OK on success).

9.5.2.16 Argus_SetCalibrationCrosstalkSequenceAmplitudeThreshold()

```
status_t Argus_SetCalibrationCrosstalkSequenceAmplitudeThreshold (
            argus_hnd_t * hnd,
            uq12_4_t value )
```

Sets the max. amplitude threshold for the crosstalk calibration sequence.

The maximum amplitude threshold defines a maximum crosstalk vector amplitude before causing an error message. If the crosstalk is too high, there is usually an issue with the measurement setup, i.e. there is still a measurement signal detected.

Parameters

hnd	The API handle; contains all internal states and data.
value	The new crosstalk calibration sequence maximum amplitude threshold value in UQ12.4 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.17 Argus_SetCalibrationCrosstalkSequenceSampleCount()

```
status_t Argus_SetCalibrationCrosstalkSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t value )
```

Sets the sample count for the crosstalk calibration sequence.

Parameters

hnd	The API handle; contains all internal states and data.
value	The new crosstalk calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.5.2.18 Argus_SetCalibrationCrosstalkVectorTable()

```
status_t Argus_SetCalibrationCrosstalkVectorTable (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Sets the custom crosstalk vector table to a specified device.

The crosstalk vectors are subtracted from the raw sampling data in the data evaluation phase.
The crosstalk vector table is a three dimensional array of type xtalk_t.
The dimensions are:

- size(0) = ARGUS_DFM_FRAME_COUNT (Dual-frequency mode A- or B-frame)
- size(1) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(2) = ARGUS_PIXELS_Y (Pixel count in y-direction)

Its recommended to use the built-in crosstalk calibration sequence (see Argus_ExecuteXtalkCalibrationSequence) to determine the crosstalk vector table.
If a constant table for all device needs to be incorporated into the sources, the Argus_GetExternalCrosstalkVectorTable_Callback should be used.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new crosstalk vector table.

Returns

Returns the status (STATUS_OK on success).

9.5.2.19 Argus_SetCalibrationGlobalRangeOffset()

```
status_t Argus_SetCalibrationGlobalRangeOffset (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            q9_22_t value )
```

Sets the global range offset value to a specified device.
The global range offset is subtracted from the raw range values.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new global range offset in meter and Q9.22 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.20 Argus_SetCalibrationPixelRangeOffsets()

```
status_t Argus_SetCalibrationPixelRangeOffsets (
            argus_hnd_t * hnd,
            argus_mode_t mode,
            q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y] )
```

Sets the relative pixel offset table to a specified device.
The relative pixel offset values are subtracted from the raw range values for each individual pixel. Note that a global range offset is applied additionally. The relative pixel offset values are meant to be with respect to the average range of all pixels, i.e. the average of all relative offsets should be 0!
The crosstalk vector table is a two dimensional array of type q0_15_t.
The dimensions are:

- size(0) = ARGUS_PIXELS_X (Pixel count in x-direction)
- size(1) = ARGUS_PIXELS_Y (Pixel count in y-direction)

Its recommended to use the built-in pixel offset calibration sequence (see Argus_ExecuteRelativeRangeOffsetCalibrationSequence) to determine the offset table for the current device.
If a constant offset table for all device needs to be incorporated into the sources, the Argus_GetExternalPixelRangeOffsets_Callback should be used.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
value	The new relative range offset in meter and Q0.15 format.

Returns

Returns the status (STATUS_OK on success).

9.5.2.21 Argus_SetCalibrationRangeOffsetSequenceSampleCount()

```
status_t Argus_SetCalibrationRangeOffsetSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t value )
```

Sets the sample count for the range offset calibration sequence.

Parameters

hnd	The API handle; contains all internal states and data.
value	The new range offset calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.5.2.22 Argus_SetCalibrationVsubSequenceSampleCount()

```
status_t Argus_SetCalibrationVsubSequenceSampleCount (
            argus_hnd_t * hnd,
            uint16_t value )
```

Sets the sample count for the substrate voltage calibration sequence.

Parameters

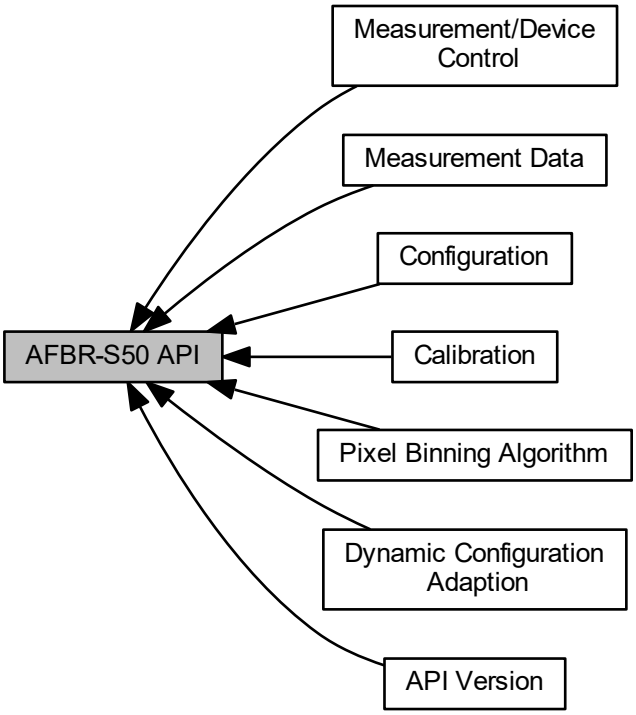
hnd	The API handle; contains all internal states and data.
value	The new substrate voltage calibration sequence sample count.

Returns

Returns the status (STATUS_OK on success).

9.6 AFBR-S50 API

The main module of the API from the AFBR-S50 SDK.
Collaboration diagram for AFBR-S50 API:



Modules

- Configuration

  Device configuration parameter definitions and API functions.
- Calibration

  Device calibration parameter definitions and API functions.
- Dynamic Configuration Adaption

  Dynamic Configuration Adaption (DCA) parameter definitions and API functions.
- Measurement/Device Control

  Measurement/Device control module.
- Pixel Binning Algorithm

  Pixel Binning Algorithm (PBA) parameter definitions and API functions.
- Measurement Data

  Measurement results data structures.
- API Version

  API and library core version number.

Macros

- #define ARGUS_PHASECOUNT 4U

Maximum number of phases per measurement cycle.

- #define ARGUS_PIXELS_X 8U

  The device pixel field size in x direction (long edge).

- #define ARGUS_PIXELS_Y 4U

  The device pixel field size in y direction (short edge).

- #define ARGUS_PIXELS ((ARGUS_PIXELS_X)*(ARGUS_PIXELS_Y))

  The total device pixel count.

- #define ARGUS_MODE_COUNT (2)

  The number of measurement modes with distinct configuration and calibration records.

Typedefs

- typedef struct Argus_Handle argus_hnd_t
- typedef int32_t s2pi_slave_t
- typedef status_t(* argus_callback_t) (status_t status, void *data)

  Generic API callback function.

Enumerations

- enum argus_module_version_t {
  MODULE_NONE = 0,
  AFBR_S50MV85G_V1 = 1,
  AFBR_S50MV85G_V2 = 2,
  AFBR_S50MV85G_V3 = 7,
  AFBR_S50LV85D_V1 = 3,
  AFBR_S50MV68B_V1 = 4,
  AFBR_S50MV85I_V1 = 5,
  AFBR_S50SV85K_V1 = 6,
  Reserved = 0b111111 }

  The AFBR-S50 module types.

- enum argus_laser_type_t {
  LASER_NONE = 0,
  LASER_H_V1 = 1,
  LASER_H_V2 = 2,
  LASER_R_V1 = 3 }

  The AFBR-S50 laser configurations.

- enum argus_chip_version_t {
  ADS0032_NONE = 0,
  ADS0032_V1_0 = 1,
  ADS0032_V1_1 = 2,
  ADS0032_V1_2 = 3 }

  The AFBR-S50 chip versions.

- enum argus_mode_t {
  ARGUS_MODE_A = 1,
  ARGUS_MODE_B = 2 }

  The measurement modes.

Functions

- status_t Argus_Init (argus_hnd_t *hnd, s2pi_slave_t spi_slave)

  Initializes the API modules and the device with default parameters.

- status_t Argus_Reinit (argus_hnd_t *hnd)

  Reinitializes the API modules and the device with default parameters.

- status_t Argus_Deinit (argus_hnd_t *hnd)

  Deinitializes the API modules and the device.

- argus_hnd_t * Argus_CreateHandle (void)

  Creates a new device data handle object to store all internal states.
- void Argus_DestroyHandle (argus_hnd_t *hnd)

  Destroys a given device data handle object.
- uint32_t Argus_GetAPIVersion (void)

  Gets the version number of the current API library.
- char const * Argus_GetBuildNumber (void)

  Gets the build number of the current API library.
- argus_module_version_t Argus_GetModuleVersion (argus_hnd_t *hnd)

  Gets the version/variant of the module.
- argus_chip_version_t Argus_GetChipVersion (argus_hnd_t *hnd)

  Gets the version number of the chip.
- argus_laser_type_t Argus_GetLaserType (argus_hnd_t *hnd)

  Gets the type number of the device laser.
- uint32_t Argus_GetChipID (argus_hnd_t *hnd)

  Gets the unique identification number of the chip.
- s2pi_slave_t Argus_GetSPISlave (argus_hnd_t *hnd)

  Gets the SPI hardware slave identifier.

9.6.1 Detailed Description

The main module of the API from the AFBR-S50 SDK.
General API for the AFBR-S50 time-of-flight sensor device family.
Include files

9.6.2 Macro Definition Documentation

9.6.2.1 ARGUS_MODE_COUNT

```
#define ARGUS_MODE_COUNT (2)
```

The number of measurement modes with distinct configuration and calibration records.

9.6.2.2 ARGUS_PHASECOUNT

```
#define ARGUS_PHASECOUNT 4U
```

Maximum number of phases per measurement cycle.
The actual phase number is defined in the register configuration. However the software does only support a fixed value of 4 yet.

9.6.2.3 ARGUS_PIXELS

```
#define ARGUS_PIXELS ((ARGUS_PIXELS_X)*(ARGUS_PIXELS_Y))
```

The total device pixel count.

9.6.2.4 ARGUS_PIXELS_X

```
#define ARGUS_PIXELS_X 8U
```

The device pixel field size in x direction (long edge).

9.6.2.5 ARGUS_PIXELS_Y

```
#define ARGUS_PIXELS_Y 4U
```

The device pixel field size in y direction (short edge).

9.6.3 Typedef Documentation

9.6.3.1 argus_callback_t

```
typedef status_t(* argus_callback_t) (status_t status, void *data)
```

Generic API callback function.

Invoked by the API. The content of the abstract data pointer depends upon the context.

Parameters

status	The module status that caused the callback. STATUS_OK if everything was as expected.
data	An abstract pointer to an user defined data. This will usually be passed to the function that also takes the callback as an parameter. Otherwise it has a special meaning such as configuration or calibration data.

Returns

Returns the status (STATUS_OK on success).

9.6.3.2 argus_hnd_t

```
typedef struct Argus_Handle argus_hnd_t
```

The data structure for the API representing a AFBR-S50 device instance.

9.6.3.3 s2pi_slave_t

```
typedef int32_t s2pi_slave_t
```

The S2PI slave identifier.

9.6.4 Enumeration Type Documentation

9.6.4.1 argus_chip_version_t

```
enum argus_chip_version_t
```

The AFBR-S50 chip versions.

Enumerator

ADS0032_NONE	No device connected or not recognized.
ADS0032_V1_0	ADS0032 v1.0
ADS0032_V1_1	ADS0032 v1.1
ADS0032_V1_2	ADS0032 v1.2

9.6.4.2 argus_laser_type_t

```
enum argus_laser_type_t
```

The AFBR-S50 laser configurations.

Enumerator

LASER_NONE	No laser connected.

Enumerator

LASER_H_V1	850nm Infra-Red VCSEL v1
LASER_H_V2	850nm Infra-Red VCSEL v2
LASER_R_V1	680nm Red VCSEL v1

9.6.4.3 argus_mode_t

`enum argus_mode_t`

The measurement modes.

Enumerator

ARGUS_MODE_A	Measurement Mode A: Long Range Mode.
ARGUS_MODE_B	Measurement Mode B: Short Range Mode.

9.6.4.4 argus_module_version_t

`enum argus_module_version_t`

The AFBR-S50 module types.

Enumerator

MODULE_NONE	No device connected or not recognized.
AFBR_S50MV85G_V1	AFBR-S50MV85G: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for medium range 3D applications. Version 1 - legacy version!
AFBR_S50MV85G_V2	AFBR-S50MV85G: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for medium range 3D applications. Version 2 - legacy version!
AFBR_S50MV85G_V3	AFBR-S50MV85G: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for medium range 3D applications. Version 7 - current version!
AFBR_S50LV85D_V1	AFBR-S50LV85D: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for long range 1D applications. Version 1 - current version!
AFBR_S50MV68B_V1	AFBR-S50MV68B: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and red, 680 nm, laser source for medium range 1D applications. Version 1 - current version!
AFBR_S50MV85I_V1	AFBR-S50MV85I: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for medium range 3D applications. Version 1 - current version!
AFBR_S50SV85K_V1	AFBR-S50MV85G: an ADS0032 based multi-pixel range finder device w/ 4x8 pixel matrix and infra-red, 850 nm, laser source for short range 3D applications. Version 1 - current version!
Reserved	Reserved for future extensions.

9.6.5 Function Documentation

9.6.5.1 Argus_CreateHandle()

```
argus_hnd_t* Argus_CreateHandle (
            void  )
```

Creates a new device data handle object to store all internal states.

The function must be called to obtain a new device handle object. The handle is basically an abstract object in memory that contains all the internal states and settings of the API module. The handle is passed to all the API methods in order to address the specified device. This allows to use the API with more than a single measurement device.

The handler is created by calling the memory allocation method from the standard library:

```
void * malloc(size_t size)
```

In order to implement an individual memory allocation method, define and implement the following weakly binded method and return a pointer to the newly allocated memory. *

```
void * Argus_Malloc (size_t size)
```

Also see the Argus_DestroyHandle method for the corresponding deallocation of the allocated memory.

Returns

Returns a pointer to the newly allocated device handler object. Returns a null pointer if the allocation failed!

Examples

01_simple_example.c, and 02_advanced_example.c.

9.6.5.2 Argus_Deinit()

```
status_t Argus_Deinit (
            argus_hnd_t * hnd )
```

Deinitializes the API modules and the device.

The function deinitializes the device and clear all internal states. Can be used to cleanup before releaseing the memory. The device can not be used any more and must be initialized again prior to next usage.

Note that the Argus_Init function must be called first! Otherwise, the function will return an error if it is called for an yet uninitialized device/handle.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

9.6.5.3 Argus_DestroyHandle()

```
void Argus_DestroyHandle (
            argus_hnd_t * hnd )
```

Destroys a given device data handle object.

The function can be called to free the previously created device data handle object in order to save memory when the device is not used any more.

Please refer to the Argus_CreateHandle method for the corresponding allocation of the memory.

The handler is destroyed by freeing the corresponding memory with the method from the standard library,

```
void free(void * ptr)
```

. In order to implement an individual memory deallocation method, define and implement the following weakly binded method and free the memory object passed to the method by a pointer.

```
void Argus_Free (void * ptr)
```

Parameters

hnd	The device handle object to be deallocated.

9.6.5.4 Argus_GetAPIVersion()

```
uint32_t Argus_GetAPIVersion (
            void  )
```

Gets the version number of the current API library.

Generic API

The version is compiled of a major (a), minor (b) and bugfix (c) number: a.b.c.
The values are encoded into a 32-bit value:

- [31 .. 24] - Major Version Number
- [23 .. 16] - Minor Version Number
- [15 .. 0] - Bugfix Version Number

To obtain the parts from the returned uin32_t value:

```
uint32_t value = Argus_GetAPIVersion();
uint8_t a = (value » 24) & 0xFFU;
uint8_t b = (value » 16) & 0xFFU;
uint8_t c = value & 0xFFFFU;
```

Returns

Returns the current version number.

9.6.5.5 Argus_GetBuildNumber()

```
char const* Argus_GetBuildNumber (
            void  )
```

Gets the build number of the current API library.

Returns

Returns the current build number as a C-string.

9.6.5.6 Argus_GetChipID()

```
uint32_t Argus_GetChipID (
            argus_hnd_t * hnd )
```

Gets the unique identification number of the chip.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the unique identification number.

9.6.5.7 Argus_GetChipVersion()

```
argus_chip_version_t Argus_GetChipVersion (
            argus_hnd_t * hnd )
```

Gets the version number of the chip.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the current version number.

9.6.5.8 Argus_GetLaserType()

```
argus_laser_type_t Argus_GetLaserType (
            argus_hnd_t * hnd )
```

Gets the type number of the device laser.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the current device laser type number.

9.6.5.9 Argus_GetModuleVersion()

```
argus_module_version_t Argus_GetModuleVersion (
            argus_hnd_t * hnd )
```

Gets the version/variant of the module.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the current module number.

9.6.5.10 Argus_GetSPISlave()

```
s2pi_slave_t Argus_GetSPISlave (
            argus_hnd_t * hnd )
```

Gets the SPI hardware slave identifier.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

The SPI hardware slave identifier.

9.6.5.11 Argus_Init()

```
status_t Argus_Init (
            argus_hnd_t * hnd,
            s2pi_slave_t spi_slave )
```

Initializes the API modules and the device with default parameters.

The function that needs to be called once after power up to initialize the modules state (i.e. the corresponding handle) and the dedicated Time-of-Flight device. In order to obtain a handle, reference the Argus_CreateHandle method.

Prior to calling the function, the required peripherals (i.e. S2PI, GPIO w/ IRQ and Timers) must be initialized and ready to use.

The function executes the following tasks:

- Initialization of the internal state represented by the handle object.
- Setup the device such that an safe configuration is present in the registers.
- Initialize sub modules such as calibration or measurement modules.

The modules configuration is initialized with reasonable default values.

Parameters

hnd	The API handle; contains all internal states and data.
spi_slave	The SPI hardware slave, i.e. the specified CS and IRQ lines. This is actually just a number that is passed to the SPI interface to distinct for multiple SPI slave devices. Note that the slave must be not equal to 0, since is reserved for error handling.

Returns

Returns the status (STATUS_OK on success).

Examples

01_simple_example.c, and 02_advanced_example.c.

9.6.5.12 Argus_Reinit()

```
status_t Argus_Reinit (
            argus_hnd_t * hnd )
```

Reinitializes the API modules and the device with default parameters.

The function reinitializes the device with default configuration. Can be used as reset sequence for the device. See Argus_Init for more information on the initialization.

Note that the Argus_Init function must be called first! Otherwise, the function will return an error if it is called for an yet uninitialized device/handle.

Parameters

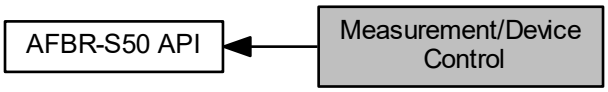
hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

9.7 Measurement/Device Control

Measurement/Device control module.
Collaboration diagram for Measurement/Device Control:



Data Structures

- struct argus_meas_frame_t

  The device measurement configuration structure.

Macros

- #define ARGUS_RAW_DATA_VALUES 132U
- #define ARGUS_RAW_DATA_SIZE (3U ∗ ARGUS_RAW_DATA_VALUES)
- #define ARGUS_AUX_CHANNEL_COUNT (5U)
- #define ARGUS_AUX_DATA_SIZE (3U ∗ ARGUS_AUX_CHANNEL_COUNT)

Functions

- status_t Argus_StartMeasurementTimer (argus_hnd_t ∗hnd, argus_callback_t cb)

  Starts the timer based measurement cycle asynchronously.
- status_t Argus_StopMeasurementTimer (argus_hnd_t ∗hnd)

  Stops the timer based measurement cycle.
- status_t Argus_TriggerMeasurement (argus_hnd_t ∗hnd, argus_callback_t cb)

  Triggers a single measurement frame asynchronously.
- status_t Argus_Abort (argus_hnd_t ∗hnd)

  Stops the currently ongoing measurements and SPI activity immediately.
- status_t Argus_GetStatus (argus_hnd_t ∗hnd)

  Checks the state of the device/driver.
- status_t Argus_Ping (argus_hnd_t ∗hnd)

  Tests the connection to the device by sending a ping message.
- status_t Argus_EvaluateData (argus_hnd_t ∗hnd, argus_results_t ∗res, void ∗raw)

  Evaluate useful information from the raw measurement data.
- status_t Argus_ExecuteXtalkCalibrationSequence (argus_hnd_t ∗hnd, argus_mode_t mode)

  Executes a crosstalk calibration measurement.
- status_t Argus_ExecuteRelativeRangeOffsetCalibrationSequence (argus_hnd_t ∗hnd, argus_mode_t mode)

  Executes a relative range offset calibration measurement.
- status_t Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence (argus_hnd_t ∗hnd, argus_mode_t mode, q9_22_t targetRange)

  Executes an absolute range offset calibration measurement.

9.7.1 Detailed Description

Measurement/Device control module.
Measurement/Device Operation
This module contains measurement and device control specific definitions and methods.

9.7.2 Macro Definition Documentation

9.7.2.1 ARGUS_AUX_CHANNEL_COUNT

```
#define ARGUS_AUX_CHANNEL_COUNT (5U)
```

The number channels for auxiliary measurements readout.

9.7.2.2 ARGUS_AUX_DATA_SIZE

```
#define ARGUS_AUX_DATA_SIZE (3U * ARGUS_AUX_CHANNEL_COUNT)
```

Size of the auxiliary data in bytes.

9.7.2.3 ARGUS_RAW_DATA_SIZE

```
#define ARGUS_RAW_DATA_SIZE (3U * ARGUS_RAW_DATA_VALUES)
```

Size of the raw data in bytes.

9.7.2.4 ARGUS_RAW_DATA_VALUES

```
#define ARGUS_RAW_DATA_VALUES 132U
```

Number of raw data values.

9.7.3 Function Documentation

9.7.3.1 Argus_Abort()

```
status_t Argus_Abort (
            argus_hnd_t * hnd )
```

Stops the currently ongoing measurements and SPI activity immediately.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

9.7.3.2 Argus_EvaluateData()

```
status_t Argus_EvaluateData (
            argus_hnd_t * hnd,
            argus_results_t * res,
            void * raw )
```

Evaluate useful information from the raw measurement data.

This function is called with a pointer to the raw results obtained from the measurement cycle. It evaluates this data and creates useful information from it. Furthermore, calibration is applied to the data. Finally, the results are used in order to adapt the device configuration to the ambient conditions in order to achieve optimal device performance. Therefore, it consists of the following sub-functions:

- Apply pre-calibration: Applies calibration steps before evaluating the data, i.e. calculations that are to the integration results directly.
- Evaluate data: Calculates measurement parameters such as range, amplitude or ambient light intensity, depending on the configurations.

- Apply post-calibration: Applies calibrations after evaluation of measurement data, i.e. calibrations applied to the calculated values such as range.
- Dynamic Configuration Adaption: checks if the configuration needs to be adjusted before the next measurement cycle in order to achieve optimum performance. Note that the configuration might not applied directly but before the next measurement starts. This is due to the fact that the device could be busy measuring already the next frame and thus no SPI activity is allowed.

However, if the device is idle, the configuration will be written immediately.

Parameters

hnd	The API handle; contains all internal states and data.
res	A pointer to the results structure that will be populated with evaluated data.
raw	The pointer to the raw data that has been obtained by the measurement finished callback.

Returns

Returns the status (STATUS_OK on success).

Examples

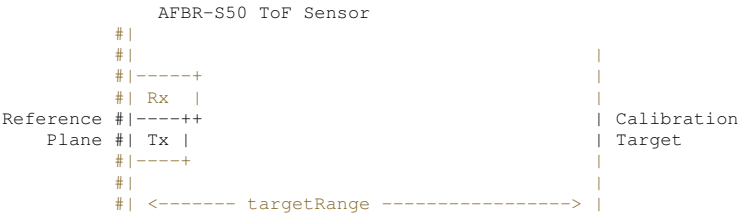
01_simple_example.c, and 02_advanced_example.c.

9.7.3.3 Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence()

```
status_t Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence (
             argus_hnd_t * hnd,
             argus_mode_t mode,
             q9_22_t targetRange )
```

Executes an absolute range offset calibration measurement.

This function immediately triggers an absolute range offset calibration measurement sequence. The ordinary measurement activity is suspended while the calibration is ongoing.
In order to perform a relative range offset calibration, a flat calibration target must be setup perpendicular to the sensors field-of-view.



There are two options to run the offset calibration: relative and absolute.

- Relative (Argus_ExecuteRelativeRangeOffsetCalibrationSequence): when the absolute distance is not essential or the distance to the calibration target is not known, the relative method can be used to compensate the relative pixel range offset w.r.t. the average range. The absolute or global range offset is not changed.
- Absolute (Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence): when the absolute distance is essential and the distance to the calibration target is known, the absolute method can be used to calibrate the absolute measured distance. Additionally, the relative pixel offset w.r.t. the average range is also compensated.

```
After calibration has finished successfully, the obtained data is
applied immediately and can be read from the API using the
#Argus_GetCalibrationPixelRangeOffsets or
#Argus_GetCalibrationGlobalRangeOffset function.
```

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.
targetRange	The absolute range between the reference plane and the calibration target in meter an Q9.22 format.

Returns

Returns the status (STATUS_OK on success).

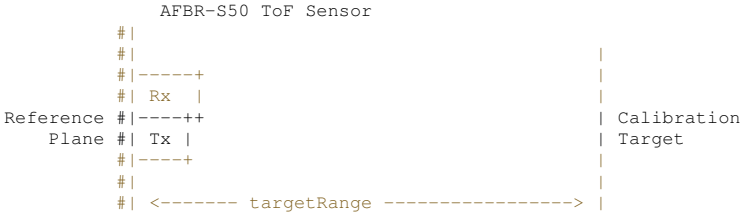
9.7.3.4 Argus_ExecuteRelativeRangeOffsetCalibrationSequence()

```
status_t Argus_ExecuteRelativeRangeOffsetCalibrationSequence (
            argus_hnd_t * hnd,
            argus_mode_t mode )
```

Executes a relative range offset calibration measurement.

This function immediately triggers a relative range offset calibration measurement sequence. The ordinary measurement activity is suspended while the calibration is ongoing.

In order to perform a relative range offset calibration, a flat calibration target must be setup perpendicular to the sensors field-of-view.



There are two options to run the offset calibration: relative and absolute.

- Relative (Argus_ExecuteRelativeRangeOffsetCalibrationSequence): when the absolute distance is not essential or the distance to the calibration target is not known, the relative method can be used to compensate the relative pixel range offset w.r.t. the average range. The absolute or global range offset is not changed.
- Absolute (Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence): when the absolute distance is essential and the distance to the calibration target is known, the absolute method can be used to calibrate the absolute measured distance. Additionally, the relative pixel offset w.r.t. the average range is also compensated.

```
After calibration has finished successfully, the obtained data is
applied immediately and can be read from the API using the
#Argus_GetCalibrationPixelRangeOffsets or
#Argus_GetCalibrationGlobalRangeOffset function.
```

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.7.3.5 Argus_ExecuteXtalkCalibrationSequence()

```
status_t Argus_ExecuteXtalkCalibrationSequence (
```

```
            argus_hnd_t * hnd,
            argus_mode_t mode )
```

Executes a crosstalk calibration measurement.

This function immediately triggers a crosstalk vector calibration measurement sequence. The ordinary measurement activity is suspended while the calibration is ongoing.
In order to perform a crosstalk calibration, the reflection of the transmitted signal must be kept from the receiver side, by either covering the TX completely (or RX respectively) or by setting up an absorbing target at far distance.
After calibration has finished successfully, the obtained data is applied immediately and can be read from the API using the Argus_GetCalibrationCrosstalkVectorTable function.

Parameters

hnd	The API handle; contains all internal states and data.
mode	The targeted measurement mode.

Returns

Returns the status (STATUS_OK on success).

9.7.3.6 Argus_GetStatus()

```
status_t Argus_GetStatus (
            argus_hnd_t * hnd )
```

Checks the state of the device/driver.

Returns the current module state:

```
    Status:
    - Idle/OK: Device and SPI interface are idle (== #STATUS_IDLE).
        - Busy: Device or SPI interface are busy (== #STATUS_BUSY).
        - Initializing: The modules and devices are currently initializing
                        (== #STATUS_INITIALIZING).
        .

        Error:
        - Not Initialized: The modules (or any submodule) has not been
                           initialized yet (== #ERROR_NOT_INITIALIZED).
        - Not Connected: No device has been connected (or connection errors
                         have occured) (== #ERROR_ARGUS_NOT_CONNECTED).
        - Timeout: A previous frame measurement has not finished within a
                   specified time (== #ERROR_TIMEOUT).
    .
```

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

Examples

01_simple_example.c.

9.7.3.7 Argus_Ping()

```
status_t Argus_Ping (
            argus_hnd_t * hnd )
```

Tests the connection to the device by sending a ping message.

A ping is transfered to the device in order to check the device and SPI connection status. Returns STATUS_OK on success and ERROR_ARGUS_NOT_CONNECTED elsewise.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

9.7.3.8 Argus_StartMeasurementTimer()

```
status_t Argus_StartMeasurementTimer (
            argus_hnd_t * hnd,
            argus_callback_t cb )
```

Starts the timer based measurement cycle asynchronously.

This function starts a timer based measurement cycle asynchronously. in the background. A periodic timer interrupt triggers the measurement frames on the ASIC and the data readout afterwards. When the frame is finished, a callback (which is passed as a parameter to the function) is invoked in order to inform the main thread to call the data evaluation method. This call is mandatory to release the data buffer for the next measurement cycle and it must not be invoked from the callback since it is within an interrupt service routine. Rather a flag should inform the main thread to invoke the evaluation as soon as possible in order to not introduce any unwanted delays to the next measurement frame. The next measurement frame will be started as soon as the pre- conditions are meet. These are:

1. timer flag set (i.e. a certain time has passed since the last measurement in order to fulfill eye-safety),
2. device idle (i.e. no measurement currently ongoing) and
3. data buffer ready (i.e. the previous data has been evaluated). Usually, the device idle and data buffer ready conditions are met before the timer tick occurs and thus the timer dictates the frame rate.

```
The callback function pointer will be invoked when the measurement
frame has finished successfully or whenever an error, that cannot
be handled internally, occurs.

The periodic timer interrupts are used to check the measurement status
```

for timeouts. An error is invoked when a measurement cycle have not finished within the specified time.

Use Argus_StopMeasurementTimer to stop the measurements.

Note

In order to use this function, the periodic interrupt timer module (see Timer: Hardware Timer Interface) must be implemented!

Parameters

hnd	The API handle; contains all internal states and data.
cb	Callback function that will be invoked when the measurement is completed. Its parameters are the status and a pointer to the results structure. If an error occurred, the status differs from STATUS_OK and the second parameter is null.

Returns

Returns the status (STATUS_OK on success).

Examples

02_advanced_example.c.

9.7.3.9 Argus_StopMeasurementTimer()

```
status_t Argus_StopMeasurementTimer (
            argus_hnd_t * hnd )
```

Stops the timer based measurement cycle.

This function stops the ongoing timer based measurement cycles that have been started using the Argus_StartMeasurementTimer function.

Parameters

hnd	The API handle; contains all internal states and data.

Returns

Returns the status (STATUS_OK on success).

9.7.3.10 Argus_TriggerMeasurement()

```
status_t Argus_TriggerMeasurement (
            argus_hnd_t * hnd,
            argus_callback_t cb )
```

Triggers a single measurement frame asynchronously.

This function immediately triggers a single measurement frame asynchronously if all the pre-conditions are met. Otherwise it returns with a corresponding status. When the frame is finished, a callback (which is passed as a parameter to the function) is invoked in order to inform the main thread to call the data evaluation method. This call is mandatory to release the data buffer for the next measurement and it must not be invoked from the callback since it is within an interrupt service routine. Rather a flag should inform the main thread to invoke the evaluation. The pre-conditions for starting a measurement frame are:

1. timer flag set (i.e. a certain time has passed since the last measurement in order to fulfill eye-safety),
2. device idle (i.e. no measurement currently ongoing) and
3. data buffer ready (i.e. the previous data has been evaluated).

   The callback function pointer will be invoked when the measurement frame has finished successfully or whenever an error, that cannot be handled internally, occurs.

The successful finishing of the measurement frame is not checked for timeouts! Instead, the user can call the Argus_GetStatus() function on a regular function to do so.

Parameters

hnd	The API handle; contains all internal states and data.
cb	Callback function that will be invoked when the measurement is completed. Its parameters are the status and a pointer to the results structure. If an error occurred, the status differs from STATUS_OK and the second parameter is null.

Returns

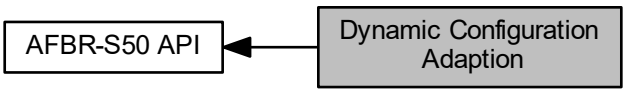
Returns the status (STATUS_OK on success).

Examples

01_simple_example.c.

9.8 Dynamic Configuration Adaption

Dynamic Configuration Adaption (DCA) parameter definitions and API functions.
Collaboration diagram for Dynamic Configuration Adaption:



Data Structures

- struct argus_cfg_dca_t

  Dynamic Configuration Adaption (DCA) Parameters.

Macros

- #define ARGUS_CFG_DCA_ATH_MIN (1U << 6U)
- #define ARGUS_CFG_DCA_ATH_MAX (0xFFFFU)
- #define ARGUS_CFG_DCA_PXTH_MIN (1U)
- #define ARGUS_CFG_DCA_PXTH_MAX (33U)
- #define ARGUS_CFG_DCA_DEPTH_MAX ((uq10_6_t)(ADS_SEQCT_N_MASK << (6U - ADS_SEQCT↩ _N_SHIFT)))
- #define ARGUS_CFG_DCA_DEPTH_MIN ((uq10_6_t)(1U))
- #define ARGUS_CFG_DCA_POWER_MAX_LSB (ADS_LASET_VCSEL_HC1_MASK >> ADS_LASET_↩ VCSEL_HC1_SHIFT)
- #define ARGUS_CFG_DCA_POWER_MIN_LSB (1)
- #define ARGUS_CFG_DCA_POWER_MAX (ADS0032_HIGH_CURRENT_LSB2MA(ARGUS_CFG_DCA_POWER_MAX_LSB + 1))
- #define ARGUS_CFG_DCA_POWER_MIN (1)
- #define ARGUS_DCA_GAIN_STAGE_COUNT (4U)
- #define ARGUS_STATE_DCA_GAIN_MASK (0x03U)
- #define ARGUS_STATE_DCA_GAIN_SHIFT (14U)
- #define ARGUS_STATE_DCA_GAIN_GET(state) (((state) >> ARGUS_STATE_DCA_GAIN_SHIFT) & ARGUS_STATE_DCA_GAIN_MASK)
- #define ARGUS_DCA_POWER_STAGE_COUNT (4U)
- #define ARGUS_STATE_DCA_POWER_MASK (0x03U)
- #define ARGUS_STATE_DCA_POWER_SHIFT (12U)
- #define ARGUS_STATE_DCA_POWER_GET(state) (((state) >> ARGUS_STATE_DCA_POWER_SHIFT) & ARGUS_STATE_DCA_POWER_MASK)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_MASK (0x0FU)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT (8U)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_GET(state) (((state) >> ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT) & ARGUS_STATE_DCA_DEPTH_SHFT_MASK)

Enumerations

- enum argus_dca_enable_t {
  DCA_ENABLE_OFF = 0,
  DCA_ENABLE_DYNAMIC = 1,
  DCA_ENABLE_STATIC = -1 }

The dynamic configuration algorithm enable flags.

- enum argus_dca_power_t {
  DCA_POWER_LOW = 0,
  DCA_POWER_MEDIUM_LOW = 1,
  DCA_POWER_MEDIUM_HIGH = 2,
  DCA_POWER_HIGH = 3 }

  The dynamic configuration algorithm Optical Output Power stages enumerator.
- enum argus_dca_gain_t {
  DCA_GAIN_LOW = 0,
  DCA_GAIN_MEDIUM_LOW = 1,
  DCA_GAIN_MEDIUM_HIGH = 2,
  DCA_GAIN_HIGH = 3 }

  The dynamic configuration algorithm Pixel Input Gain stages enumerator.
- enum argus_state_t {
  ARGUS_STATE_NONE = 0,
  ARGUS_STATE_MEASUREMENT_MODE = 1U << 0U,
  ARGUS_STATE_DUAL_FREQ_MODE = 1U << 1U,
  ARGUS_STATE_MEASUREMENT_FREQ = 1U << 2U,
  ARGUS_STATE_DEBUG_MODE = 1U << 3U,
  ARGUS_STATE_GOLDEN_PIXEL_MODE = 1U << 4U,
  ARGUS_STATE_BGL_WARNING = 1U << 5U,
  ARGUS_STATE_BGL_ERROR = 1U << 6U,
  ARGUS_STATE_PLL_LOCKED = 1U << 7U,
  ARGUS_STATE_DCA_POWER_LOW = DCA_GAIN_LOW << ARGUS_STATE_DCA_POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_MED_LOW = DCA_GAIN_MEDIUM_LOW << ARGUS_STATE_DCA_↩
  POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_MED_HIGH = DCA_GAIN_MEDIUM_HIGH << ARGUS_STATE_DCA_↩
  POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_HIGH = DCA_GAIN_HIGH << ARGUS_STATE_DCA_POWER_SHIFT,
  ARGUS_STATE_DCA_GAIN_LOW = DCA_GAIN_LOW << ARGUS_STATE_DCA_GAIN_SHIFT,
  ARGUS_STATE_DCA_GAIN_MED_LOW = DCA_GAIN_MEDIUM_LOW << ARGUS_STATE_DCA_GAI↩
  N_SHIFT,
  ARGUS_STATE_DCA_GAIN_MED_HIGH = DCA_GAIN_MEDIUM_HIGH << ARGUS_STATE_DCA_G↩
  AIN_SHIFT,
  ARGUS_STATE_DCA_GAIN_HIGH = DCA_GAIN_HIGH << ARGUS_STATE_DCA_GAIN_SHIFT }

  State flags for the current frame.

9.8.1 Detailed Description

Dynamic Configuration Adaption (DCA) parameter definitions and API functions.

The DCA contains an algorithms that detect ambient conditions and adopt the device configuration to the changing parameters dynamically while operating the sensor. This is achieved by rating the currently received signal quality and changing the device configuration accordingly to the gathered information from the current measurement frame results before the next integration cycle starts.
The DCA consists of the following features:

- Static or Dynamic mode. The first is utilizing the nominal values while the latter is dynamically adopting between min. and max. value and starting from the nominal values.
- Analog Integration Depth Adaption (from multiple patterns down to single pulses)
- Optical Output Power Adaption
- Pixel Input Gain Adaption (w/ ambient light rejection)
- ADC Sensitivity (i.e. ADC Range) Adaption
- Power Saving Ratio (to decrease the average output power and thus the current consumption)
- All that features are heeding the Laser Safety limits.

9.8.2 Macro Definition Documentation

9.8.2.1 ARGUS_CFG_DCA_ATH_MAX

```
#define ARGUS_CFG_DCA_ATH_MAX (0xFFFFU)
```
The maximum amplitude threshold value.

9.8.2.2 ARGUS_CFG_DCA_ATH_MIN

```
#define ARGUS_CFG_DCA_ATH_MIN (1U << 6U)
```
The minimum amplitude threshold value.

9.8.2.3 ARGUS_CFG_DCA_DEPTH_MAX

```
#define ARGUS_CFG_DCA_DEPTH_MAX ((uq10_6_t)(ADS_SEQCT_N_MASK << (6U - ADS_SEQCT_N_SHIFT)))
```
The maximum analog integration depth in UQ10.6 format, i.e. the maximum pattern count per sample.

9.8.2.4 ARGUS_CFG_DCA_DEPTH_MIN

```
#define ARGUS_CFG_DCA_DEPTH_MIN ((uq10_6_t)(1U))
```
The minimum analog integration depth in UQ10.6 format, i.e. the minimum pattern count per sample.

9.8.2.5 ARGUS_CFG_DCA_POWER_MAX

```
#define ARGUS_CFG_DCA_POWER_MAX (ADS0032_HIGH_CURRENT_LSB2MA(ARGUS_CFG_DCA_POWER_MAX_LSB + 1))
```
The maximum optical output power, i.e. the maximum VCSEL 1 high current in LSB.

9.8.2.6 ARGUS_CFG_DCA_POWER_MAX_LSB

```
#define ARGUS_CFG_DCA_POWER_MAX_LSB (ADS_LASET_VCSEL_HC1_MASK >> ADS_LASET_VCSEL_HC1_SHIFT)
```
The maximum optical output power, i.e. the maximum VCSEL 1 high current in LSB.

9.8.2.7 ARGUS_CFG_DCA_POWER_MIN

```
#define ARGUS_CFG_DCA_POWER_MIN (1)
```
The minimum optical output power, i.e. the minimum VCSEL 1 high current in mA.

9.8.2.8 ARGUS_CFG_DCA_POWER_MIN_LSB

```
#define ARGUS_CFG_DCA_POWER_MIN_LSB (1)
```
The minimum optical output power, i.e. the minimum VCSEL 1 high current in mA.

9.8.2.9 ARGUS_CFG_DCA_PXTH_MAX

```
#define ARGUS_CFG_DCA_PXTH_MAX (33U)
```
The maximum saturated pixel threshold value.

9.8.2.10 ARGUS_CFG_DCA_PXTH_MIN

```
#define ARGUS_CFG_DCA_PXTH_MIN (1U)
```
The minimum saturated pixel threshold value.

9.8.2.11 ARGUS_DCA_GAIN_STAGE_COUNT

```
#define ARGUS_DCA_GAIN_STAGE_COUNT (4U)
```
The dynamic configuration algorithm Pixel Input Gain stage count.

9.8.2.12 ARGUS_DCA_POWER_STAGE_COUNT

```
#define ARGUS_DCA_POWER_STAGE_COUNT (4U)
```
The dynamic configuration algorithm Optical Output Power stage count.

9.8.2.13 ARGUS_STATE_DCA_DEPTH_SHFT_GET

```
#define ARGUS_STATE_DCA_DEPTH_SHFT_GET(
            state ) (((state) >> ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT) & ARGUS_STATE_DCA_DEPTH_SHFT_MASK)
```
Getter for the dynamic configuration algorithm Max. Analog Integration Depth shift value.

9.8.2.14 ARGUS_STATE_DCA_DEPTH_SHFT_MASK

```
#define ARGUS_STATE_DCA_DEPTH_SHFT_MASK (0x0FU)
```
The dynamic configuration algorithm state mask for the Max. Analog Integration Depth shift value.

9.8.2.15 ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT

```
#define ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT (8U)
```
The dynamic configuration algorithm state mask for the Max. Analog Integration Depth shift value.

9.8.2.16 ARGUS_STATE_DCA_GAIN_GET

```
#define ARGUS_STATE_DCA_GAIN_GET(
            state ) (((state) >> ARGUS_STATE_DCA_GAIN_SHIFT) & ARGUS_STATE_DCA_GAIN_MASK)
```
Getter for the dynamic configuration algorithm Pixel Input Gain stage.

9.8.2.17 ARGUS_STATE_DCA_GAIN_MASK

```
#define ARGUS_STATE_DCA_GAIN_MASK (0x03U)
```
The dynamic configuration algorithm state mask for the Pixel Input Gain stage.

9.8.2.18 ARGUS_STATE_DCA_GAIN_SHIFT

```
#define ARGUS_STATE_DCA_GAIN_SHIFT (14U)
```
The dynamic configuration algorithm state mask for the Pixel Input Gain stage.

9.8.2.19 ARGUS_STATE_DCA_POWER_GET

```
#define ARGUS_STATE_DCA_POWER_GET(
            state ) (((state) >> ARGUS_STATE_DCA_POWER_SHIFT) & ARGUS_STATE_DCA_POWER_MASK)
```
Getter for the dynamic configuration algorithm Optical Output Power stage.

9.8.2.20 ARGUS_STATE_DCA_POWER_MASK

```
#define ARGUS_STATE_DCA_POWER_MASK (0x03U)
```
The dynamic configuration algorithm state mask for the Optical Output Power stage.

9.8.2.21 ARGUS_STATE_DCA_POWER_SHIFT

```
#define ARGUS_STATE_DCA_POWER_SHIFT (12U)
```
The dynamic configuration algorithm state mask for the Optical Output Power stage.

9.8.3 Enumeration Type Documentation

9.8.3.1 argus_dca_enable_t

enum argus_dca_enable_t

The dynamic configuration algorithm enable flags.

Enumerator

DCA_ENABLE_OFF	DCA is disabled and will be completely skipped.
DCA_ENABLE_DYNAMIC	DCA is enabled and will dynamically adjust the device configuration.
DCA_ENABLE_STATIC	DCA is enabled and will apply the static (nominal) values to the device.

9.8.3.2 argus_dca_gain_t

`enum argus_dca_gain_t`

The dynamic configuration algorithm Pixel Input Gain stages enumerator.

Enumerator

DCA_GAIN_LOW	Low gain stage.
DCA_GAIN_MEDIUM_LOW	Medium low gain stage.
DCA_GAIN_MEDIUM_HIGH	Medium high gain stage.
DCA_GAIN_HIGH	High gain stage.

9.8.3.3 argus_dca_power_t

`enum argus_dca_power_t`

The dynamic configuration algorithm Optical Output Power stages enumerator.

Enumerator

DCA_POWER_LOW	Low output power stage.
DCA_POWER_MEDIUM_LOW	Medium low output power stage.
DCA_POWER_MEDIUM_HIGH	Medium high output power stage.
DCA_POWER_HIGH	High output power stage.

9.8.3.4 argus_state_t

`enum argus_state_t`

State flags for the current frame.

State flags determine the current state of the measurement frame:

- [0]: ARGUS_STATE_MEASUREMENT_MODE: Measurement Mode:
  - 0: Mode A
  - 1: Mode B
- [1]: ARGUS_STATE_DUAL_FREQ_MODE: Dual Frequency Mode Enabled Flag
  - 0: Disabled, measurements w/ base frequency only
  - 1: Enabled, measurements w/ detuned frequency
- [2]: ARGUS_STATE_MEASUREMENT_FREQ: Measurement Frequency for Dual Frequency Mode, (only valid if ARGUS_STATE_DUAL_FREQ_MODE flag is set)
  - 0: A-Frame w/ detuned frequency
  - 1: B-Frame w/ detuned frequency

- [3]: ARGUS_STATE_DEBUG_MODE
- [4]: ARGUS_STATE_GOLDEN_PIXEL_MODE
- [5]: ARGUS_STATE_BGL_WARNING
- [6]: ARGUS_STATE_BGL_ERROR
- [7]: ARGUS_STATE_PLL_LOCKED
  - 0: PLL_LOCKED bit was not set at start of integration;
  - 0: PLL_LOCKED bit was set at start of integration;
- [8-11]: Max. Depth Shift Value
- [12-13]: Power Stages
- [14-15]: Gain Stages

Enumerator

ARGUS_STATE_NONE	No state flag set.
ARGUS_STATE_MEASUREMENT_MODE	0x01: Measurement Mode. • 0: Mode A: Long Range / Medium Precision • 1: Mode B: Short Range / High Precision
ARGUS_STATE_DUAL_FREQ_MODE	0x02: Dual Frequency Mode Enabled. • 0: Disabled: measurements with base frequency, • 1: Enabled: measurement with detuned frequency.
ARGUS_STATE_MEASUREMENT_FREQ	0x04: Measurement Frequency for Dual Frequency Mode (only if ARGUS_STATE_DUAL_FREQ_MODE flag is set). • 0: A-Frame w/ detuned frequency, • 1: B-Frame w/ detuned frequency
ARGUS_STATE_DEBUG_MODE	0x08: Debug Mode. If set, the range value of erroneous pixels are not cleared or reset. • 0: Disabled (default). • 1: Enabled.
ARGUS_STATE_GOLDEN_PIXEL_MODE	0x10: Golden Pixel Mode Flag. Set whenever the Pixel Binning Algorithm is operating in the Golden Pixel Mode. • 0: Normal Pixel Binning Mode. • 1: Golden Pixel Mode.
ARGUS_STATE_BGL_WARNING	0x20: Background Light Warning Flag. Set whenever the background light is very high and the measurement data might be unreliable. • 0: No Warning Background Light is within valid range. • 1: Warning: Background Light is very high.

Enumerator

ARGUS_STATE_BGL_ERROR	0x40: Background Light Error Flag. Set whenever the background light is too high and the measurement data is unreliable or invalid. • 0: No Error, Background Light is within valid range. • 1: Error: Background Light is too high.
ARGUS_STATE_PLL_LOCKED	0x80: PLL_LOCKED bit. • 0: PLL not locked at start of integration. • 1: PLL locked at start of integration.
ARGUS_STATE_DCA_POWER_LOW	DCA is in low Optical Output Power stage.
ARGUS_STATE_DCA_POWER_MED_LOW	DCA is in medium-low Optical Output Power stage.
ARGUS_STATE_DCA_POWER_MED_HIGH	DCA is in medium-high Optical Output Power stage.
ARGUS_STATE_DCA_POWER_HIGH	DCA is in high Optical Output Power stage.
ARGUS_STATE_DCA_GAIN_LOW	DCA is in low Pixel Input Gain stage.
ARGUS_STATE_DCA_GAIN_MED_LOW	DCA is in medium-low Pixel Input Gain stage.
ARGUS_STATE_DCA_GAIN_MED_HIGH	DCA is in medium-high Pixel Input Gain stage.
ARGUS_STATE_DCA_GAIN_HIGH	DCA is in high Pixel Input Gain stage.

9.9 Dual Frequency Mode

Dual Frequency Mode (DFM) parameter definitions and API functions.

Macros

- #define ARGUS_DFM_FRAME_COUNT (2U)
- #define ARGUS_DFM_MODE_COUNT (2U)

Enumerations

- enum argus_dfm_mode_t {
  DFM_MODE_OFF = 0U,
  DFM_MODE_4X = 1U,
  DFM_MODE_8X = 2U }

9.9.1 Detailed Description

Dual Frequency Mode (DFM) parameter definitions and API functions.

The DFM is an algorithm to extend the unambiguous range of the sensor by utilizing two detuned measurement frequencies.

The AFBR-S50 API provides three measurement modes:

- 1X: Single Frequency Measurement
- 4X: Dual Frequency Measurement w/ 4 times the unambiguous range of the Single Frequency Measurement
- 8X: Dual Frequency Measurement w/ 8 times the unambiguous range of the Single Frequency Measurement

9.9.2 Macro Definition Documentation

9.9.2.1 ARGUS_DFM_FRAME_COUNT

```
#define ARGUS_DFM_FRAME_COUNT (2U)
```

The Dual Frequency Mode frequency count.

9.9.2.2 ARGUS_DFM_MODE_COUNT

```
#define ARGUS_DFM_MODE_COUNT (2U)
```

The Dual Frequency Mode measurement modes count. Excluding the disabled mode.

9.9.3 Enumeration Type Documentation

9.9.3.1 argus_dfm_mode_t

```
enum argus_dfm_mode_t
```

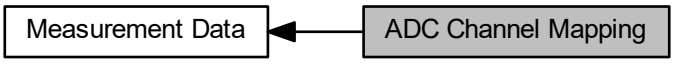
The Dual Frequency Mode measurement modes enumeration.

Enumerator

DFM_MODE_OFF	Single Frequency Measurement Mode (w/ 1x Unambiguous Range).
DFM_MODE_4X	4X Dual Frequency Measurement Mode (w/ 4x Unambiguous Range).
DFM_MODE_8X	8X Dual Frequency Measurement Mode (w/ 8x Unambiguous Range).

9.10 ADC Channel Mapping

Pixel ADC Channel (n) to x-y-Index Mapping.
Collaboration diagram for ADC Channel Mapping:



Macros

- #define PIXEL_XY2N(x, y) ((((x) ^ 7) << 1) | ((y) & 2) << 3 | ((y) & 1))

  Macro to determine the channel number of an specified Pixel.
- #define PIXEL_N2X(n) ((((n) >> 1U) & 7) ^ 7)

  Macro to determine the x index of an specified Pixel channel.
- #define PIXEL_N2Y(n) (((n) & 1U) | (((n) >> 3) & 2U))

  Macro to determine the y index of an specified Pixel channel.
- #define PIXELN_ISENABLED(msk, ch) (((msk) >> (ch)) & 0x01U)

  Macro to determine if a ADC Pixel channel was enabled from a pixel mask.
- #define PIXELN_ENABLE(msk, ch) ((msk) |= (0x01U << (ch)))

  Macro enables an ADC Pixel channel in a pixel mask.
- #define PIXELN_DISABLE(msk, ch) ((msk) &= (~(0x01U << (ch))))

  Macro disables an ADC Pixel channel in a pixel mask.
- #define PIXELXY_ISENABLED(msk, x, y) (PIXELN_ISENABLED(msk, PIXEL_XY2N(x, y)))

  Macro to determine if an ADC Pixel channel was enabled from a pixel mask.
- #define PIXELXY_ENABLE(msk, x, y) (PIXELN_ENABLE(msk, PIXEL_XY2N(x, y)))

  Macro enables an ADC Pixel channel in a pixel mask.
- #define PIXELXY_DISABLE(msk, x, y) (PIXELN_DISABLE(msk, PIXEL_XY2N(x, y)))

  Macro disables an ADC Pixel channel in a pixel mask.
- #define CHANNELN_ISENABLED(msk, ch) (((msk) >> ((ch) - 32U)) & 0x01U)

  Macro to determine if a ADC channel was enabled from a channel mask.
- #define CHANNELN_ENABLE(msk, ch) ((msk) |= (0x01U << ((ch) - 32U)))

  Macro to determine if a ADC channel was enabled from a channel mask.
- #define CHANNELN_DISABLE(msk, ch) ((msk) &= (~(0x01U << ((ch) - 32U))))

  Macro to determine if a ADC channel was enabled from a channel mask.
- #define PIXEL_COUNT(pxmsk) popcount(pxmsk)

  Macro to determine the number of enabled pixel channels via a popcount algorithm.
- #define CHANNEL_COUNT(pxmsk, chmsk) (popcount(pxmsk) + popcount(chmsk))

  Macro to determine the number of enabled channels via a popcount algorithm.

9.10.1 Detailed Description

Pixel ADC Channel (n) to x-y-Index Mapping.
The ADC Channels of each pixel or auxiliary channel on the device is numbered in a way that is convenient on the chip. The macros in this module are defined in order to obtain the x-y-indices of each channel and vice versa.

9.10.2 Macro Definition Documentation

9.10.2.1 CHANNEL_COUNT

```
#define CHANNEL_COUNT(
            pxmsk,
            chmsk ) (popcount(pxmsk) + popcount(chmsk))
```

Macro to determine the number of enabled channels via a popcount algorithm.

Parameters

pxmsk	32-bit pixel mask
chmsk	32-bit channel mask

Returns

The count of enabled ADC channels.

9.10.2.2 CHANNELN_DISABLE

```
#define CHANNELN_DISABLE(
            msk,
            ch ) ((msk) &= (~(0x01U << ((ch) - 32U))))
```

Macro to determine if a ADC channel was enabled from a channel mask.

Parameters

msk	32-bit channel mask
ch	channel number of the ADC channel.

Returns

True if the ADC channel n was enabled, false elsewise.

9.10.2.3 CHANNELN_ENABLE

```
#define CHANNELN_ENABLE(
            msk,
            ch ) ((msk) |= (0x01U << ((ch) - 32U)))
```

Macro to determine if a ADC channel was enabled from a channel mask.

Parameters

msk	32-bit channel mask
ch	channel number of the ADC channel.

Returns

True if the ADC channel n was enabled, false elsewise.

9.10.2.4 CHANNELN_ISENABLED

```
#define CHANNELN_ISENABLED(
            msk,
            ch ) (((msk) >> ((ch) - 32U)) & 0x01U)
```

Macro to determine if a ADC channel was enabled from a channel mask.

Parameters

msk	32-bit channel mask
ch	channel number of the ADC channel.

Returns

True if the ADC channel n was enabled, false elsewise.

9.10.2.5 PIXEL_COUNT

```
#define PIXEL_COUNT(
            pxmsk ) popcount(pxmsk)
```

Macro to determine the number of enabled pixel channels via a popcount algorithm.

Parameters

pxmsk	32-bit pixel mask

Returns

The count of enabled pixel channels.

9.10.2.6 PIXEL_N2X

```
#define PIXEL_N2X(
            n ) ((((n) >> 1U) & 7) ^ 7)
```

Macro to determine the x index of an specified Pixel channel.

Parameters

n	The channel number of the pixel.

Returns

The x index number of the pixel.

9.10.2.7 PIXEL_N2Y

```
#define PIXEL_N2Y(
            n ) (((n) & 1U) | (((n) >> 3) & 2U))
```

Macro to determine the y index of an specified Pixel channel.

Parameters

n	The channel number of the pixel.

Returns

The y index number of the pixel.

9.10.2.8 PIXEL_XY2N

```
#define PIXEL_XY2N(
              x,
              y ) ((((x) ^ 7) << 1) | ((y) & 2) << 3 | ((y) & 1))
```

Macro to determine the channel number of an specified Pixel.

Parameters

x	The x index of the pixel.
y	The y index of the pixel.

Returns

The channel number n of the pixel.

9.10.2.9 PIXELN_DISABLE

```
#define PIXELN_DISABLE(
              msk,
              ch ) ((msk) &= (~(0x01U << (ch))))
```

Macro disables an ADC Pixel channel in a pixel mask.

Parameters

msk	The 32-bit pixel mask
ch	The channel number of the pixel.

9.10.2.10 PIXELN_ENABLE

```
#define PIXELN_ENABLE(
              msk,
              ch ) ((msk) |= (0x01U << (ch)))
```

Macro enables an ADC Pixel channel in a pixel mask.

Parameters

msk	The 32-bit pixel mask
ch	The channel number of the pixel.

9.10.2.11 PIXELN_ISENABLED

```
#define PIXELN_ISENABLED(
              msk,
              ch ) (((msk) >> (ch)) & 0x01U)
```

Macro to determine if a ADC Pixel channel was enabled from a pixel mask.

Parameters

msk	The 32-bit pixel mask
ch	The channel number of the pixel.

Returns

True if the pixel channel n was enabled, false elsewise.

9.10.2.12 PIXELXY_DISABLE

```
#define PIXELXY_DISABLE(
              msk,
              x,
              y ) (PIXELN_DISABLE(msk, PIXEL_XY2N(x, y)))
```

Macro disables an ADC Pixel channel in a pixel mask.

Parameters

msk	32-bit pixel mask
x	x index of the pixel.
y	y index of the pixel.

9.10.2.13 PIXELXY_ENABLE

```
#define PIXELXY_ENABLE(
              msk,
              x,
              y ) (PIXELN_ENABLE(msk, PIXEL_XY2N(x, y)))
```

Macro enables an ADC Pixel channel in a pixel mask.

Parameters

msk	32-bit pixel mask
x	x index of the pixel.
y	y index of the pixel.

9.10.2.14 PIXELXY_ISENABLED

```
#define PIXELXY_ISENABLED(
              msk,
              x,
              y ) (PIXELN_ISENABLED(msk, PIXEL_XY2N(x, y)))
```

Macro to determine if an ADC Pixel channel was enabled from a pixel mask.

Parameters

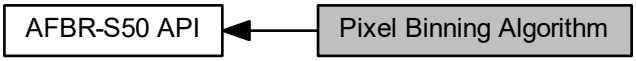
msk	32-bit pixel mask
x	x index of the pixel.
y	y index of the pixel.

Returns

True if the pixel (x,y) was enabled, false elsewise.

9.11 Pixel Binning Algorithm

Pixel Binning Algorithm (PBA) parameter definitions and API functions.
Collaboration diagram for Pixel Binning Algorithm:



Data Structures

- struct argus_cfg_pba_t

  The pixel binning algorithm settings data structure.

Enumerations

- enum argus_pba_flags_t {
  PBA_ENABLE = 1U << 0U,
  PBA_ENABLE_GOLDPX = 1U << 5U,
  PBA_ENABLE_MIN_DIST_SCOPE = 1U << 6U }

  Enable flags for the pixel binning algorithm.
- enum argus_pba_averaging_mode_t {
  PBA_SIMPLE_AVG = 1U,
  PBA_LINEAR_AMPLITUDE_WEIGHTED_AVG = 2U }

  The averaging modes for the pixel binning algorithm.

9.11.1 Detailed Description

Pixel Binning Algorithm (PBA) parameter definitions and API functions.
Defines the generic pixel binning algorithm (PBA) setup parameters and data structure.
The PBA module contains filter algorithms that determine the pixels with the best signal quality and extract an 1d distance information from the filtered pixels.
The pixel filter algorithm is a three-stage filter with a fallback value:

1. A fixed pre-filter mask is applied to statically disable specified pixels.

2. A relative and absolute amplitude filter is applied in the second stage. The relative filter is determined by a ratio of the maximum amplitude off all available (i.e. not filtered in stage 1) pixels. Pixels that have an amplitude below the relative threshold are dismissed. The same holds true for the absolute amplitude threshold. All pixel with smaller amplitude are dismissed.
   The relative threshold is useful to setup a distance measurement scenario. All well illuminated pixels are selected and considered for the final 1d distance. The absolute threshold is used to dismiss pixels that are below the noise level. The latter would be considered for the 1d result if the maximum amplitude is already very low.

3. A distance filter is used to distinguish pixels that target the actual object from pixels that see the brighter background, e.g. white walls. Thus, the pixel with the minimum distance is referenced and all pixel that have a distance between the minimum and the given minimum distance scope are selected for the 1d distance result. The minimum distance scope is determined by an relative (to the current minimum distance) and an absolute value. The larger scope value is the relevant one, i.e. the relative distance scope can be used to heed the increasing noise at larger distances.

4. If all of the above filters fail to determine a single valid pixel, the golden pixel is used as a fallback value. The golden pixel is the pixel that sits right at the focus point of the optics at large distances.

After filtering is done, there may be more than a single pixel left to determine the 1d signal. Therefore several averaging methods are implemented to obtain the best 1d result from many pixels. See argus_pba_averaging_mode_t for details.

9.11.2 Enumeration Type Documentation

9.11.2.1 argus_pba_averaging_mode_t

`enum argus_pba_averaging_mode_t`

The averaging modes for the pixel binning algorithm.

Enumerator

PBA_SIMPLE_AVG	Evaluate the 1D range from all available pixels using a simple average.
PBA_LINEAR_AMPLITUDE_WEIGHTED_AVG	Evaluate the 1D range from all available pixels using a linear amplitude weighted averaging method. Formula: x_mean = sum(x_i ∗ A_i) / sum(A_i)

9.11.2.2 argus_pba_flags_t

`enum argus_pba_flags_t`

Enable flags for the pixel binning algorithm.

Determines the pixel binning algorithm feature enable status.

- [0]: PBA_ENABLE: Enables the pixel binning feature.
- [1]: reserved
- [2]: reserved
- [3]: reserved
- [4]: reserved
- [5]: PBA_ENABLE_GOLDPX: Enables the golden pixel feature.
- [6]: PBA_ENABLE_MIN_DIST_SCOPE: Enables the minimum distance scope feature.
- [7]: reserved

Enumerator

PBA_ENABLE	Enables the pixel binning feature.
PBA_ENABLE_GOLDPX	Enables the golden pixel.
PBA_ENABLE_MIN_DIST_SCOPE	Enables the minimum distance scope filter.

9.12 Measurement Data

Measurement results data structures.
Collaboration diagram for Measurement Data:

Modules

- ADC Channel Mapping

  Pixel ADC Channel (n) to x-y-Index Mapping.

Data Structures

- struct argus_pixel_t

  The evaluated measurement results per pixel.
- struct argus_results_bin_t

  The 1d measurement results data structure.
- struct argus_results_aux_t

  The auxiliary measurement results data structure.
- struct argus_results_t

  The measurement results data structure.

Macros

- #define ARGUS_AMPLITUDE_MAX (0xFFF0U)
- #define ARGUS_RANGE_MAX (Q9_22_MAX)

Enumerations

- enum argus_px_status_t {
  PIXEL_OK = 0,
  PIXEL_OFF = 1U << 0U,
  PIXEL_SAT = 1U << 1U,
  PIXEL_BIN_EXCL = 1U << 2U,
  PIXEL_AMPL_MIN = 1U << 3U,
  PIXEL_PREFILTERED = 1U << 4U,
  PIXEL_NO_SIGNAL = 1U << 5U,
  PIXEL_OUT_OF_SYNC = 1U << 6U,
  PIXEL_STALLED = 1U << 7U }

  Status flags for the evaluated pixel structure.

9.12.1 Detailed Description

Measurement results data structures.
The interface defines all data structures that correspond to the AFBR-S50 measurement results, e.g.

- 1D distance and amplitude values,
- 3D distance and amplitude values (i.e. per pixel),
- Auxiliary channel measurement results (VDD, IAPD, temperature, ...)

- Device and result status
- ...

9.12.2 Macro Definition Documentation

9.12.2.1 ARGUS_AMPLITUDE_MAX

```
#define ARGUS_AMPLITUDE_MAX (0xFFF0U)
```

Maximum amplitude value in UQ12.4 format.

9.12.2.2 ARGUS_RANGE_MAX

```
#define ARGUS_RANGE_MAX (Q9_22_MAX)
```

Maximum range value in Q9.22 format. Also used as a special value to determine no object detected or infinity range.

9.12.3 Enumeration Type Documentation

9.12.3.1 argus_px_status_t

```
enum argus_px_status_t
```

Status flags for the evaluated pixel structure.

Determines the pixel status. 0 means OK (PIXEL_OK).

- [0]: PIXEL_OFF: Pixel was disabled and not read from the device.
- [1]: PIXEL_SAT: The pixel was saturated.
- [2]: PIXEL_BIN_EXCL: The pixel was excluded from the 1D result.
- [3]: PIXEL_AMPL_MIN: The pixel amplitude has evaluated to 0.
- [4]: PIXEL_PREFILTERED: The was pre-filtered by static mask.
- [5]: PIXEL_NO_SIGNAL: The pixel has no valid signal.
- [6]: PIXEL_OUT_OF_SYNC: The pixel has lost signal trace.
- [7]: PIXEL_STALLED: The pixel value is stalled due to errors.

Enumerator

PIXEL_OK	0x00: Pixel status OK.
PIXEL_OFF	0x01: Pixel is disabled (in hardware) and no data has been read from the device.
PIXEL_SAT	0x02: Pixel is saturated (i.e. at least one saturation bit for any sample is set or the sample is in the invalidity area).
PIXEL_BIN_EXCL	0x04: Pixel is excluded from the pixel binning (1d) result.
PIXEL_AMPL_MIN	0x08: Pixel amplitude minimum underrun (i.e. the amplitude calculation yields 0).
PIXEL_PREFILTERED	0x10: Pixel is pre-filtered by the static pixel binning pre-filter mask, i.e. the pixel is disabled by software.
PIXEL_NO_SIGNAL	0x20: Pixel amplitude is below its threshold value. The received signal strength is too low to evaluate a valid signal. The range value is set to the maximum possible value (approx. 512 m).
PIXEL_OUT_OF_SYNC	0x40: Pixel is not in sync with respect to the dual frequency algorithm. I.e. the pixel may have a correct value but is estimated into the wrong unambiguous window.

Enumerator

PIXEL_STALLED	0x80: Pixel is stalled due to one of the following reasons: • PIXEL_SAT • PIXEL_AMPL_MIN • PIXEL_OUT_OF_SYNC • Global Measurement Error A stalled pixel does not update its measurement data and keeps the previous values. If the issue is resolved, the stall disappears and the pixel is updating again.

9.13 Shot Noise Monitor

Shot Noise Monitor (SNM) parameter definitions and API functions.

Enumerations

- enum argus_snm_mode_t {
  SNM_MODE_STATIC_INDOOR = 0U,
  SNM_MODE_STATIC_OUTDOOR = 1U,
  SNM_MODE_DYNAMIC = 2U }

9.13.1 Detailed Description

Shot Noise Monitor (SNM) parameter definitions and API functions.
The SNM is an algorithm to monitor and react on shot noise induced by harsh environment conditions like high ambient light.
The AFBR-S50 API provides three modes:

- Dynamic: Automatic mode, automatically adopts to current ambient conditions.
- Static (Outdoor): Static mode, optimized for outdoor applications.
- Static (Indoor): Static mode, optimized for indoor applications.

9.13.2 Enumeration Type Documentation

9.13.2.1 argus_snm_mode_t

`enum argus_snm_mode_t`
The Shot Noise Monitor modes enumeration.

Enumerator

SNM_MODE_STATIC_INDOOR	Static Shot Noise Monitoring Mode, optimized for indoor applications. Assumes the best case scenario, i.e. no bad influence from ambient conditions. Thus it uses a fixed setting that will result in the best performance. Equivalent to Shot Noise Monitoring disabled.
SNM_MODE_STATIC_OUTDOOR	Static Shot Noise Monitoring Mode, optimized for outdoor applications. Assumes the worst case scenario, i.e. it uses a fixed setting that will work under all ambient conditions.
SNM_MODE_DYNAMIC	Dynamic Shot Noise Monitoring Mode. Adopts the system performance dynamically to the current ambient conditions.

9.14 Status Codes

Status and Error Code Definitions.

Typedefs

- typedef int32_t status_t

  Type used for all status and error return values.

Enumerations

- enum Status {
  STATUS_OK = 0,
  STATUS_IDLE = 0,
  STATUS_IGNORE = 1,
  STATUS_BUSY = 2,
  STATUS_INITIALIZING = 3,
  ERROR_FAIL = -1,
  ERROR_ABORTED = -2,
  ERROR_READ_ONLY = -3,
  ERROR_OUT_OF_RANGE = -4,
  ERROR_INVALID_ARGUMENT = -5,
  ERROR_TIMEOUT = -6,
  ERROR_NOT_INITIALIZED = -7,
  ERROR_NOT_SUPPORTED = -8,
  ERROR_NOT_IMPLEMENTED = -9,
  STATUS_S2PI_GPIO_MODE = 51,
  ERROR_S2PI_RX_ERROR = -51,
  ERROR_S2PI_TX_ERROR = -52,
  ERROR_S2PI_INVALID_STATE = -53,
  ERROR_S2PI_INVALID_BAUD_RATE = -54,
  ERROR_S2PI_INVALID_SLAVE = -55,
  ERROR_NVM_INVALID_FILE_VERSION = -98,
  ERROR_NVM_OUT_OF_RANGE = -99,
  STATUS_ARGUS_BUFFER_BUSY = 104,
  STATUS_ARGUS_POWERLIMIT = 105,
  STATUS_ARGUS_UNDERFLOW = 107,
  STATUS_ARGUS_NO_OBJECT = 108,
  STATUS_ARGUS_EEPROM_BIT_ERROR = 109,
  STATUS_ARGUS_INVALID_EEPROM = 110,
  ERROR_ARGUS_NOT_CONNECTED = -101,
  ERROR_ARGUS_INVALID_CFG = -102,
  ERROR_ARGUS_INVALID_MODE = -105,
  ERROR_ARGUS_BIAS_VOLTAGE_REINIT = -107,
  ERROR_ARGUS_EEPROM_FAILURE = -109,
  ERROR_ARGUS_STALLED = -110,
  ERROR_ARGUS_BGL_EXCEEDANCE = -111,
  ERROR_ARGUS_XTALK_AMPLITUDE_EXCEEDANCE = -112,
  ERROR_ARGUS_LASER_FAILURE = -113,
  ERROR_ARGUS_DATA_INTEGRITY_LOST = -114,
  ERROR_ARGUS_RANGE_OFFSET_CALIBRATION_FAILED = -115,
  ERROR_ARGUS_BUSY = -191,
  ERROR_ARGUS_UNKNOWN_MODULE = -199,
  ERROR_ARGUS_UNKNOWN_CHIP = -198,
  ERROR_ARGUS_UNKNOWN_LASER = -197,
  STATUS_ARGUS_BUSY_CFG_UPDATE = 193,
  STATUS_ARGUS_BUSY_CAL_UPDATE = 194,
  STATUS_ARGUS_BUSY_CAL_SEQ = 195,

STATUS_ARGUS_BUSY_MEAS = 196,
STATUS_ARGUS_STARTING = 100,
STATUS_ARGUS_ACTIVE = 103 }

9.14.1 Detailed Description

Status and Error Code Definitions.

Defines status and error codes for function return values. Basic status number structure:

- 0 is OK or no error.
- negative values determine errors.
- positive values determine warnings or status information.

9.14.2 Typedef Documentation

9.14.2.1 status_t

```
typedef int32_t status_t
```

Type used for all status and error return values.

Basic status number structure:

- 0 is OK or no error.
- negative values determine errors.
- positive values determine warnings or status information.

9.14.3 Enumeration Type Documentation

9.14.3.1 Status

```
enum Status
```

AFBR-S50 API status and error return codes.

Enumerator

STATUS_OK	0: Status for success/no error.
STATUS_IDLE	0: Status for device/module/hardware idle. Implies STATUS_OK.
STATUS_IGNORE	1: Status to be ignored.
STATUS_BUSY	2: Status for device/module/hardware busy.
STATUS_INITIALIZING	3: Status for device/module/hardware is currently initializing.
ERROR_FAIL	-1: Error for generic fail/error.
ERROR_ABORTED	-2: Error for process aborted by user/external.
ERROR_READ_ONLY	-3: Error for invalid read only operations.
ERROR_OUT_OF_RANGE	-4: Error for out of range parameters.
ERROR_INVALID_ARGUMENT	-5: Error for invalid argument passed to an function.
ERROR_TIMEOUT	-6: Error for timeout occurred.
ERROR_NOT_INITIALIZED	-7: Error for not initialized modules.
ERROR_NOT_SUPPORTED	-8: Error for not supported.
ERROR_NOT_IMPLEMENTED	-9: Error for yet not implemented functions.
STATUS_S2PI_GPIO_MODE	51: SPI is disabled and pins are used in GPIO mode.

Enumerator

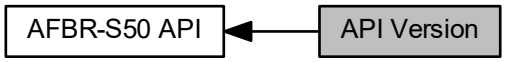
ERROR_S2PI_RX_ERROR	-51: Error occurred on the Rx line.
ERROR_S2PI_TX_ERROR	-52: Error occurred on the Tx line.
ERROR_S2PI_INVALID_STATE	-53: Called a function at a wrong driver state.
ERROR_S2PI_INVALID_BAUD_RATE	-54: The specified baud rate is not valid.
ERROR_S2PI_INVALID_SLAVE	-55: The specified slave identifier is not valid.
ERROR_NVM_INVALID_FILE_VERSION	-98: Flash Error: The version of the settings in the flash memory is not compatible.
ERROR_NVM_OUT_OF_RANGE	-99: Flash Error: The memory is out of range.
STATUS_ARGUS_BUFFER_BUSY	104: AFBR-S50 Status: All (internal) raw data buffers are currently in use. The measurement was not executed due to lack of available raw data buffers. Please call Argus_EvaluateData to free the buffers.
STATUS_ARGUS_POWERLIMIT	105: AFBR-S50 Status: The measurement was not executed/started due to output power limitations.
STATUS_ARGUS_UNDERFLOW	107: AFBR-S50 Status: No valid signal was detected at any active pixel via the Pixel Binning Algorithm. The Golden Pixel was Choosen as a fallback value that is consider to be the last pixel that has a valid signal for low reflective (or far away) objects. The current results should be considered carefully.
STATUS_ARGUS_NO_OBJECT	108: AFBR-S50 Status: No object was detected within the field-of-view and measurement range of the device.
STATUS_ARGUS_EEPROM_BIT_ERROR	109: AFBR-S50 Status: The readout algorithm for the EEPROM has detected a bit error which has been corrected. However, if more than a single bit error has occurred, the corrected value is invalid! This cannot be distinguished from the valid case. Thus, if the error starts to occur, the sensor should be replaced soon!
STATUS_ARGUS_INVALID_EEPROM	110: AFBR-S50 Status: Inconsistent EEPROM readout data. No calibration trimming values are applied. The calibration remains invalid.
ERROR_ARGUS_NOT_CONNECTED	-101: AFBR-S50 Error: No device connected. Initial SPI tests failed.
ERROR_ARGUS_INVALID_CFG	-102: AFBR-S50 Error: Inconsistent configuration parameters.
ERROR_ARGUS_INVALID_MODE	-105: AFBR-S50 Error: Invalid measurement mode configuration parameter.
ERROR_ARGUS_BIAS_VOLTAGE_REINIT	-107: AFBR-S50 Error: The APD bias voltage is reinitializing due to a dropout. The current measurement data set is invalid!
ERROR_ARGUS_EEPROM_FAILURE	-109: AFBR-S50 Error: The EEPROM readout has failed. The failure is detected by three distinct read attempts, each resulting in invalid data. Note: this state differs from that STATUS_ARGUS_EEPROM_BIT_ERROR such that it is usually temporarily and due to harsh ambient conditions.
ERROR_ARGUS_STALLED	-110: AFBR-S50 Error: The measurement signals of all active pixels are invalid and thus the 1D range is also invalid and stalled. This means the range value is not updated and kept at the previous valid value.

Enumerator

ERROR_ARGUS_BGL_EXCEEDANCE	-111: AFBR-S50 Error: The background light is too bright.
ERROR_ARGUS_XTALK_AMPLITUDE_EXCEEDANCE	-112: AFBR-S50 Error: The crosstalk vector amplitude is too high.
ERROR_ARGUS_LASER_FAILURE	-113: AFBR-S50 Error: Laser malfunction! Laser Safety may not be given!
ERROR_ARGUS_DATA_INTEGRITY_LOST	-114: AFBR-S50 Error: Register data integrity is lost (e.g. due to unexpected power-on-reset cycle or invalid write cycle of SPI. System tries to reset the values.
ERROR_ARGUS_RANGE_OFFSET_CALIBRATION_FAILED	-115: AFBR-S50 Error: The range offsets calibration failed!
ERROR_ARGUS_BUSY	-191: AFBR-S50 Error: The device is currently busy and cannot execute the requested command.
ERROR_ARGUS_UNKNOWN_MODULE	-199: AFBR-S50 Error: Unknown module number.
ERROR_ARGUS_UNKNOWN_CHIP	-198: AFBR-S50 Error: Unknown chip version number.
ERROR_ARGUS_UNKNOWN_LASER	-197: AFBR-S50 Error: Unknown laser type number.
STATUS_ARGUS_BUSY_CFG_UPDATE	193: AFBR-S50 Status (internal): The device is currently busy with updating the configuration (i.e. with writing register values).
STATUS_ARGUS_BUSY_CAL_UPDATE	194: AFBR-S50 Status (internal): The device is currently busy with updating the calibration data (i.e. writing to register values).
STATUS_ARGUS_BUSY_CAL_SEQ	195: AFBR-S50 Status (internal): The device is currently executing a calibration sequence.
STATUS_ARGUS_BUSY_MEAS	196: AFBR-S50 Status (internal): The device is currently executing a measurement cycle.
STATUS_ARGUS_STARTING	100: AFBR-S50 Status (internal): The ASIC is initializing a new measurement, i.e. a register value is written that starts an integration cycle on the ASIC.
STATUS_ARGUS_ACTIVE	103: AFBR-S50 Status (internal): The ASIC is performing an integration cycle.

9.15 API Version

API and library core version number.
Collaboration diagram for API Version:



Macros

- #define ARGUS_API_VERSION_MAJOR 1
- #define ARGUS_API_VERSION_MINOR 2
- #define ARGUS_API_VERSION_BUGFIX 3
- #define ARGUS_API_VERSION_BUILD "20201120091253"
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 24) | ((minor) << 16) | (bugfix))
- #define ARGUS_API_VERSION

9.15.1 Detailed Description

API and library core version number.
Contains the AFBR-S50 API and Library Core Version Number.

9.15.2 Macro Definition Documentation

9.15.2.1 ARGUS_API_VERSION

```
#define ARGUS_API_VERSION
```

Value:

```
MAKE_VERSION((ARGUS_API_VERSION_MAJOR), \
(ARGUS_API_VERSION_MINOR), \
(ARGUS_API_VERSION_BUGFIX))
```

Version number of the AFBR-S50 API.

9.15.2.2 ARGUS_API_VERSION_BUGFIX

```
#define ARGUS_API_VERSION_BUGFIX 3
```

Bugfix version number of the AFBR-S50 API.

9.15.2.3 ARGUS_API_VERSION_BUILD

```
#define ARGUS_API_VERSION_BUILD "20201120091253"
```

Build version nunber of the AFBR-S50 API.

9.15.2.4 ARGUS_API_VERSION_MAJOR

```
#define ARGUS_API_VERSION_MAJOR 1
```

Major version number of the AFBR-S50 API.

9.15.2.5 ARGUS_API_VERSION_MINOR

```
#define ARGUS_API_VERSION_MINOR 2
```

Minor version number of the AFBR-S50 API.

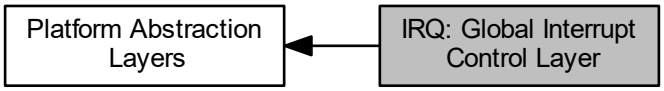
9.15.2.6 MAKE_VERSION

```
#define MAKE_VERSION(
            major,
            minor,
            bugfix ) (((major) << 24) | ((minor) << 16) | (bugfix))
```

Construct the version number for drivers.

9.16 IRQ: Global Interrupt Control Layer

Global Interrupt Control Layer.
Collaboration diagram for IRQ: Global Interrupt Control Layer:



Functions

- void IRQ_UNLOCK (void)

  Enable IRQ Interrupts.

- void IRQ_LOCK (void)

  Disable IRQ Interrupts.

9.16.1 Detailed Description

Global Interrupt Control Layer.
This module provides functionality to globally enable/disable interrupts by turning the I-bit in the CPSR on/off.
Here is a simple example implementation using the CMSIS functions "__enable_irq()" and "__disable_irq()". An integer counter is used to achieve nested interrupt disabling:

```
// Global lock level counter value.
static volatile int g_irq_lock_ct;
// Global unlock all interrupts using CMSIS function "__enable_irq()".
void IRQ_UNLOCK(void)
{
    assert(g_irq_lock_ct > 0);
    if (--g_irq_lock_ct <= 0)
    {
        g_irq_lock_ct = 0;
        __enable_irq();
    }
}
// Global lock all interrupts using CMSIS function "__disable_irq()".
void IRQ_LOCK(void)
{
    __disable_irq();
    g_irq_lock_ct++;
}
```

Note

The IRQ locking mechanism is used to create atomic sections (within the scope of the AFBR-S50 API) that are very few processor instruction only. It does NOT lock interrupts for considerable amounts of time.

The interrupts utilized by the AFBR-S50 API can be interrupted by other, higher prioritized interrupts, e.g. some system critical interrupts. In this case, the IRQ_LOCK/IRQ_UNLOCK mechanism can be implemented such that only the interrupts required for the AFBR-S50 API are locked. The above example is dedicated to a ARM Corex-M0 architecture, where interrupts can only disabled at a global scope. Other architectures like ARM Cortex-M4 allow selective disabling of interrupts.

9.16.2 Function Documentation

9.16.2.1 IRQ_LOCK()

```
void IRQ_LOCK (
            void  )
```

Disable IRQ Interrupts.

Disables IRQ interrupts by setting the I-bit in the CPSR. Can only be executed in Privileged modes.

9.16.2.2 IRQ_UNLOCK()

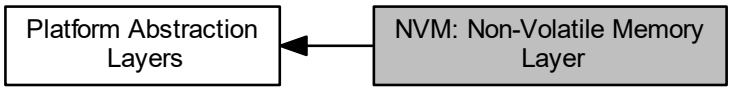
```
void IRQ_UNLOCK (
            void  )
```

Enable IRQ Interrupts.

Enables IRQ interrupts by clearing the I-bit in the CPSR. Can only be executed in Privileged modes.

9.17 NVM: Non-Volatile Memory Layer

Non-Volatile Memory Layer.
Collaboration diagram for NVM: Non-Volatile Memory Layer:



Functions

- status_t NVM_Init (uint32_t size)

  Initializes the non-volatile memory unit and reserves a chunk of memory.
- status_t NVM_Write (uint32_t offset, uint32_t size, uint8_t const ∗buf)

  Write a block of data to the non-volatile memory.
- status_t NVM_Read (uint32_t offset, uint32_t size, uint8_t ∗buf)

  Reads a block of data from the non-volatile memory.

9.17.1 Detailed Description

Non-Volatile Memory Layer.
This module provides functionality to access the non-volatile memory (e.g. flash) on the underlying platform.
This module is optional and only required if calibration data needs to be stored within the API.

Note

The implementation of this module is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disables the NVM feature.

9.17.2 Function Documentation

9.17.2.1 NVM_Init()

```
status_t NVM_Init (
            uint32_t size )
```

Initializes the non-volatile memory unit and reserves a chunk of memory.
The function is called upon API initialization sequence. If available, the non-volatile memory module reserves a chunk of memory with the provides number of bytes (size) and returns with STATUS_OK.
If not implemented, the function should return ERROR_NOT_IMPLEMENTED in oder to inform the API to not use the NVM module.
After initialization, the API calls the NVM_Write and NVM_Read methods to write within the reserved chunk of memory.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disables the NVM feature.

Parameters

size	The required size of NVM to store all parameters.

Returns

Returns the status (STATUS_OK on success).

9.17.2.2 NVM_Read()

```
status_t NVM_Read (
            uint32_t offset,
            uint32_t size,
            uint8_t * buf )
```

Reads a block of data from the non-volatile memory.

The function is called whenever the API wants to read data from the previously reserved (NVM_Init) memory block.
The data shall be read at a given offset and with a given size.
If no NVM module is available, the function can return with error ERROR_NOT_IMPLEMENTED.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disables the NVM feature.

Parameters

offset	The index offset where the first byte needs to be read.
size	The number of bytes to be read.
buf	The pointer to the data buffer to copy the data to.

Returns

Returns the status (STATUS_OK on success).

9.17.2.3 NVM_Write()

```
status_t NVM_Write (
            uint32_t offset,
            uint32_t size,
            uint8_t const * buf )
```

Write a block of data to the non-volatile memory.

The function is called whenever the API wants to write data into the previously reserved (NVM_Init) memory block.
The data shall be written at a given offset and with a given size.
If no NVM module is available, the function can return with error ERROR_NOT_IMPLEMENTED.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disables the NVM feature.

Parameters

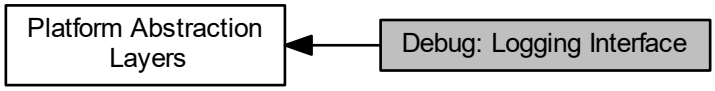
offset	The index offset where the first byte needs to be written.
size	The number of bytes to be written.
buf	The pointer to the data buffer with the data to be written.

Returns

Returns the status (STATUS_OK on success).

9.18 Debug: Logging Interface

Logging interface for the AFBR-S50 API.
Collaboration diagram for Debug: Logging Interface:



Functions

- status_t print (const char *fmt_s,...)

  A printf-like function to print formated data to an debugging interface.

9.18.1 Detailed Description

Logging interface for the AFBR-S50 API.
This interface provides logging utility functions. Defines a printf-like function that is used to print error and log messages.

9.18.2 Function Documentation

9.18.2.1 print()

```
status_t print (
            const char * fmt_s,
             ...  )
```

A printf-like function to print formated data to an debugging interface.
Writes the C string pointed by fmt_t to an output. If format includes format specifiers (subsequences beginning with %), the additional arguments following fmt_t are formatted and inserted in the resulting string replacing their respective specifiers.
To enable the print functionality, an implementation of the function must be provided that maps the output to an interface like UART or a debugging console, e.g. by forwarding to standard printf() method.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that does nothing. This will improve the performance but no error messages are logged.

The naming is different from the standard printf() on purpose to prevent builtin compiler optimizations.

Parameters

fmt_s	The usual print() format string.
...	The usual print() parameters. *

Returns

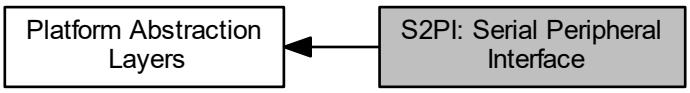
Returns the status (STATUS_OK on success).

Examples

01_simple_example.c, and 02_advanced_example.c.

9.19 S2PI: Serial Peripheral Interface

S2PI: SPI incl. GPIO Hardware Layer Module.
Collaboration diagram for S2PI: Serial Peripheral Interface:



Typedefs

- typedef status_t(∗ s2pi_callback_t) (status_t status, void ∗param)
  S2PI layer callback function type for the SPI transfer completed event.
- typedef void(∗ s2pi_irq_callback_t) (void ∗param)
  S2PI layer callback function type for the GPIO interrupt event.
- typedef int32_t s2pi_slave_t

Enumerations

- enum s2pi_pin_t {
  S2PI_CLK,
  S2PI_CS,
  S2PI_MOSI,
  S2PI_MISO,
  S2PI_IRQ }

Functions

- status_t S2PI_GetStatus (void)
  Returns the status of the SPI module.
- status_t S2PI_TransferFrame (s2pi_slave_t slave, uint8_t const ∗txData, uint8_t ∗rxData, size_t frameSize, s2pi_callback_t callback, void ∗callbackData)
  Transfers a single SPI frame asynchronously.
- status_t S2PI_Abort (void)
  Terminates a currently ongoing asynchronous SPI transfer.
- status_t S2PI_SetIrqCallback (s2pi_slave_t slave, s2pi_irq_callback_t callback, void ∗callbackData)
  Set a callback for the GPIO IRQ for a specified S2PI slave.
- uint32_t S2PI_ReadIrqPin (s2pi_slave_t slave)
  Reads the current status of the IRQ pin.
- status_t S2PI_CycleCsPin (s2pi_slave_t slave)
  Cycles the chip select line.
- status_t S2PI_CaptureGpioControl (void)
  Captures the S2PI pins for GPIO usage.
- status_t S2PI_ReleaseGpioControl (void)
  Releases the S2PI pins from GPIO usage and switches back to SPI mode.
- status_t S2PI_WriteGpioPin (s2pi_slave_t slave, s2pi_pin_t pin, uint32_t value)
  Writes the output for a specified SPI pin in GPIO mode.
- status_t S2PI_ReadGpioPin (s2pi_slave_t slave, s2pi_pin_t pin, uint32_t ∗value)
  Reads the input from a specified SPI pin in GPIO mode.

9.19.1 Detailed Description

S2PI: SPI incl. GPIO Hardware Layer Module.

The S2PI module consists of a standard SPI interface plus a single GPIO interrupt line. Furthermore, the SPI pins are accessible via GPIO control to allow a software emulation of additional protocols using the same pins.

SPI interface:

The SPI interface is based on a single function:
S2PI_TransferFrame. This function transfers a specified number of bytes via the interfaces MOSI line and simultaneously reads the incoming data on the MOSI line. The read can also be skipped. The transfer happen asynchronously, e.g. via a DMA request. After finishing the transfer, the provided callback is invoked with the status of the transfer and the provided abstract parameter. Furthermore, the functions receives a slave parameter that can be used to connect multiple slaves, each with its individual chip select line.
The interface also provides functionality to change the SPI baud rate. An additional abort method is used to cancel the ongoing transfer.

GPIO interface:

The GPIO interface handles the measurement finished interrupt from the device. When the device invokes the interrupt, it pulls the interrupt line to low. Thus the interrupt must trigger when a transition from high to low occurs on the interrupt line.
The module simply invokes a callback when this interrupt the pending. The S2PI_SetIrqCallback method is used to install the callback for a specified slave. Each slave will have its own interrupt line. An additional callback parameter can be set that would be passed to the callback function.
In addition to the interrupt, all SPI pins need to be accessible as GPIO pins through the interface. One basic operation would be to cycle the chip select pin which resets the device. Additional, the device contains an EEPROM that is connected to the SPI pins but requires a different protocol that is not compatible to any standard SPI interface. Therefore, the interface provides the possibility to switch to GPIO control that allows to emulate the EEPROM protocol via software bit banging. Two methods are provided to switch forth and back between SPI and GPIO control. In GPIO mode, several functions are used to read and write the individual GPIO pins.
Note that the GPIO mode is only required to readout the EEPROM at initialization of the device, i.e. during execution of the Argus_Init or Argus_Reinit method. The GPIO mode is not used during measurements.

9.19.2 Typedef Documentation

9.19.2.1 s2pi_callback_t

```
typedef status_t(* s2pi_callback_t) (status_t status, void *param)
```

S2PI layer callback function type for the SPI transfer completed event.

Parameters

status	The status of the completed transfer (STATUS_OK on success).
param	The provided (optional, can be null) callback parameter.

Returns

Returns the status (STATUS_OK on success).

9.19.2.2 s2pi_irq_callback_t

```
typedef void(* s2pi_irq_callback_t) (void *param)
```

S2PI layer callback function type for the GPIO interrupt event.

Parameters

param	The provided (optional, can be null) callback parameter.

9.19.2.3 s2pi_slave_t

```
typedef int32_t s2pi_slave_t
```

The S2PI slave identifier. Basically an user defined enumerable type that can be used to identify the slave within the SPI module.

9.19.3 Enumeration Type Documentation

9.19.3.1 s2pi_pin_t

```
enum s2pi_pin_t
```

The enumeration of S2PI pins.

Enumerator

S2PI_CLK	The SPI clock pin.
S2PI_CS	The SPI chip select pin.
S2PI_MOSI	The SPI MOSI pin.
S2PI_MISO	The SPI MISO pin.
S2PI_IRQ	The IRQ pin.

9.19.4 Function Documentation

9.19.4.1 S2PI_Abort()

```
status_t S2PI_Abort (
            void  )
```

Terminates a currently ongoing asynchronous SPI transfer.
When a callback is set for the current ongoing activity, it is invoked with the ERROR_ABORTED error byte.

Returns

Returns the status (STATUS_OK on success).

9.19.4.2 S2PI_CaptureGpioControl()

```
status_t S2PI_CaptureGpioControl (
            void  )
```

Captures the S2PI pins for GPIO usage.
The SPI is disabled (module status: STATUS_S2PI_GPIO_MODE) and the pins are configured for GPIO operation. The GPIO control must be release with the S2PI_ReleaseGpioControl function in order to switch back to ordinary SPI functionality.

Note

This function is only called during device initialization!

Returns

Returns the status (STATUS_OK on success).

9.19.4.3 S2PI_CycleCsPin()

```
status_t S2PI_CycleCsPin (
            s2pi_slave_t slave )
```

Cycles the chip select line.

In order to cancel the integration on the ASIC, a fast toggling of the chip select pin of the corresponding SPI slave is required. Therefore, this function toggles the CS from high to low and back. The SPI instance for the specified S2PI slave must be idle, otherwise the status STATUS_BUSY is returned.

Parameters

slave	The specified S2PI slave.

Returns

Returns the status (STATUS_OK on success).

9.19.4.4 S2PI_GetStatus()

```
status_t S2PI_GetStatus (
            void  )
```

Returns the status of the SPI module.

Returns

Returns the status:

- STATUS_IDLE: No SPI transfer or GPIO access is ongoing.
- STATUS_BUSY: An SPI transfer is in progress.
- STATUS_S2PI_GPIO_MODE: The module is in GPIO mode.

9.19.4.5 S2PI_ReadGpioPin()

```
status_t S2PI_ReadGpioPin (
            s2pi_slave_t slave,
            s2pi_pin_t pin,
            uint32_t * value )
```

Reads the input from a specified SPI pin in GPIO mode.

This function reads the value of an SPI pin if the SPI pins are captured for GPIO operation via the S2PI_CaptureGpioControl previously.

Note

This function is only called during device initialization!

Parameters

slave	The specified S2PI slave.
pin	The specified S2PI pin.
value	The GPIO pin state to read (0 = low, 1 = high).

Returns

Returns the status (STATUS_OK on success).

9.19.4.6 S2PI_ReadIrqPin()

```
uint32_t S2PI_ReadIrqPin (
            s2pi_slave_t slave )
```

Reads the current status of the IRQ pin.

In order to keep a low priority for GPIO IRQs, the state of the IRQ pin must be read in order to reliable check for chip timeouts.

The execution of the interrupt service routine for the data-ready interrupt from the corresponding GPIO pin might be delayed due to priority issues. The delayed execution might disable the timeout for the eye-safety checker too late causing false error messages. In order to overcome the issue, the state of the IRQ GPIO input pin is read before raising a timeout error in order to check if the device has already finished but the IRQ is still pending to be executed!

Parameters

slave	The specified S2PI slave.

Returns

Returns 1U if the IRQ pin is high (IRQ not pending) and 0U if the devices pulls the pin to low state (IRQ pending).

9.19.4.7 S2PI_ReleaseGpioControl()

```
status_t S2PI_ReleaseGpioControl (
            void  )
```

Releases the S2PI pins from GPIO usage and switches back to SPI mode.

The GPIO pins are configured for SPI operation and the GPIO mode is left. Must be called if the pins are captured for GPIO operation via the S2PI_CaptureGpioControl function.

Note

This function is only called during device initialization!

Returns

Returns the status (STATUS_OK on success).

9.19.4.8 S2PI_SetIrqCallback()

```
status_t S2PI_SetIrqCallback (
            s2pi_slave_t slave,
            s2pi_irq_callback_t callback,
            void * callbackData )
```

Set a callback for the GPIO IRQ for a specified S2PI slave.

Parameters

slave	The specified S2PI slave.
callback	A callback function to be invoked when the specified S2PI slave IRQ occurs. Pass a null pointer to disable the callback.
callbackData	A pointer to a state that will be passed to the callback. Pass a null pointer if not used.

Returns

Returns the status:

- STATUS_OK: Successfully installation of the callback.

- ERROR_S2PI_INVALID_SLAVE: A wrong slave identifier is provided.

9.19.4.9 S2PI_TransferFrame()

```
status_t S2PI_TransferFrame (
            s2pi_slave_t slave,
            uint8_t const * txData,
            uint8_t * rxData,
            size_t frameSize,
            s2pi_callback_t callback,
            void * callbackData )
```

Transfers a single SPI frame asynchronously.

Transfers a single SPI frame in asynchronous manner. The Tx data buffer is written to the device via the MOSI line. Optionally, the data on the MISO line is written to the provided Rx data buffer. If null, the read data is dismissed. Note that Rx and Tx buffer can be identical. I.e. the same buffer is used for writing and reading data. First, a byte is transmitted and the received byte overwrites the previously send value.
The transfer of a single frame requires to not toggle the chip select line to high in between the data frame. The maximum number of bytes transfered in a single SPI transfer is given by the data value register of the device, which is 396 data bytes plus a single address byte: 397 bytes.
An optional callback is invoked when the asynchronous transfer is finished. If the `callback` parameter is a null pointer, no callback is provided. Note that the provided buffer must not change while the transfer is ongoing.
Use the slave parameter to determine the corresponding slave via the given chip select line.
Usually, two distinct interrupts are required to handle the RX and TX ready events. The callback must be invoked from whichever interrupt comes after the SPI transfer has been finished. Note that new SPI transfers are invoked from within the callback function (i.e. from within the interrupt service routine of same priority).

Parameters

slave	The specified S2PI slave.
txData	The 8-bit values to write to the SPI bus MOSI line.
rxData	The 8-bit values received from the SPI bus MISO line (pass a null pointer if the data don't need to be read).
frameSize	The number of 8-bit values to be sent/received.
callback	A callback function to be invoked when the transfer is finished. Pass a null pointer if no callback is required.
callbackData	A pointer to a state that will be passed to the callback. Pass a null pointer if not used.

Returns

Returns the status:

- STATUS_OK: Successfully invoked the transfer.
- ERROR_INVALID_ARGUMENT: An invalid parameter has been passed.
- ERROR_S2PI_INVALID_SLAVE: A wrong slave identifier is provided.
- STATUS_BUSY: An SPI transfer is already in progress. The transfer was not started.
- STATUS_S2PI_GPIO_MODE: The module is in GPIO mode. The transfer was not started.

9.19.4.10 S2PI_WriteGpioPin()

```
status_t S2PI_WriteGpioPin (
            s2pi_slave_t slave,
            s2pi_pin_t pin,
            uint32_t value )
```

Writes the output for a specified SPI pin in GPIO mode.

This function writes the value of an SPI pin if the SPI pins are captured for GPIO operation via the S2PI_CaptureGpioControl previously.

Note

Since some GPIO peripherals switch the GPIO pins very fast a delay must be added after each GBIO access (i.e. right before returning from the S2PI_WriteGpioPin method) in order to decrease the baud rate of the software EEPROM protocol. Increase the delay if timing issues occur while reading the EERPOM. For example: Delay = 10 µsec => Baud Rate < 100 kHz

This function is only called during device initialization!

Parameters

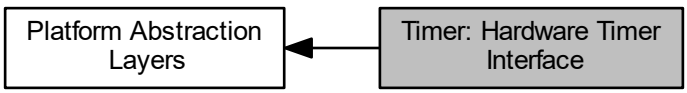
slave	The specified S2PI slave.
pin	The specified S2PI pin.
value	The GPIO pin state to write (0 = low, 1 = high).

Returns

Returns the status (STATUS_OK on success).

9.20 Timer: Hardware Timer Interface

Timer implementations for lifetime counting as well as periodic callback.
Collaboration diagram for Timer: Hardware Timer Interface:



Typedefs

- typedef void(∗ timer_cb_t) (void ∗param)

  The callback function type for periodic interrupt timer.

Functions

- void Timer_GetCounterValue (uint32_t ∗hct, uint32_t ∗lct)

  Obtains the lifetime counter value from the timers.
- status_t Timer_SetCallback (timer_cb_t f)

  Installs an periodic timer callback function.
- status_t Timer_SetInterval (uint32_t dt_microseconds, void ∗param)

  Sets the timer interval for a specified callback parameter.
- status_t Timer_Start (uint32_t dt_microseconds, void ∗param)

  Starts the timer for a specified callback parameter.
- status_t Timer_Stop (void ∗param)

  Stops the timer for a specified callback parameter.

9.20.1 Detailed Description

Timer implementations for lifetime counting as well as periodic callback.
The module provides an interface to the timing utilities that are required by the AFBR-S50 time-of-flight sensor API.
Two essential features have to be provided by the user code:

1. Time Measurement Capability: In order to keep track of outgoing signals, the API needs to measure elapsed time. In order to provide optimum device performance, the granularity should be around 10 to 100 microseconds.
2. Periodic Callback: The API provides an automatic starting of measurement cycles at a fixed frame rate via a periodic interrupt timer. If this feature is not used, implementation of the periodic interrupts can be skipped. An weak default implementation is provide in the API.

The time measurement feature is simply implemented by the function Timer_GetCounterValue. Whenever the function is called, the provided counter values must be written with the values obtained by the current time.
The periodic interrupt timer is a simple callback interface. After installing the callback function pointer via Timer_SetCallback, the timer can be started by setting interval via Timer_SetInterval or Timer_Start. From then, the callback is invoked periodically as the corresponding interval may specify. The timer is stopped via Timer_Stop or by setting the interval to 0. The interval can be updated at any time by updating the interval via the Timer_SetInterval function. To any of these functions, an abstract parameter pointer must be passed. This parameter is passed back to the callback any time it is invoked.
In order to provide the usage of multiple devices, an mechanism is introduced to allow the installation of multiple callback interval at the same time. Therefore, the abstract parameter pointer is used to identify the corresponding

callback interval. For example, there are two callbacks for two intervals, t1 and t2, required. The user can start two timers by calling the Timer_Start method twice, but with an individual parameter pointer, ptr1 and ptr2, each:

```
Timer_Start(100000, ptr1); // 10 ms callback w/ parameter ptr1
Timer_Start(200000, ptr2); // 20 ms callback w/ parameter ptr1
```

Note that the implemented timer module must therefore support as many different intervals as instances of the AFBR-S50 device are used.

9.20.2 Typedef Documentation

9.20.2.1 timer_cb_t

```
typedef void(* timer_cb_t) (void *param)
```

The callback function type for periodic interrupt timer.

The function that is invoked every time a specified interval elapses. An abstract parameter is passed to the function whenever it is called.

Parameters

param	An abstract parameter to be passed to the callback. This is also the identifier of the given interval.

9.20.3 Function Documentation

9.20.3.1 Timer_GetCounterValue()

```
void Timer_GetCounterValue (
            uint32_t * hct,
            uint32_t * lct )
```

Obtains the lifetime counter value from the timers.

The function is required to get the current time relative to any point in time, e.g. the startup time. The returned values `hct` and `lct` are given in seconds and microseconds respectively. The current elapsed time since the reference time is then calculated from:

t_now [µsec] = hct ∗ 1000000 µsec + lct ∗ 1 µsec

Note that the accuracy/granularity of the lifetime counter does not need to be 1 µsec. Usually, a granularity of approximately 100 µsec is sufficient. However, in case of very high frame rates (above 1000 frames per second), it is recommended to implement an even lower granularity (somewhere in the 10 µsec regime).

It must be guaranteed, that each call of the Timer_GetCounterValue function must provide a value that is greater or equal, but never lower, than the value returned from the previous call.

A hardware based implementation of the lifetime counter functionality would be to chain two distinct timers such that counter 2 increases its value when counter 1 wraps to 0. The easiest way is to setup counter 1 to wrap exactly every second. Counter 1 would than count the sub-seconds (i.e. µsec) value (`lct`) and counter 2 the seconds (`hct`) value. A 16-bit counter is sufficient in case of counter 1 while counter 2 must be a 32-bit version.

In case of a lack of available hardware timers, a software solution can be used that requires only a 16-bit timer. In a simple scenario, the timer is configured to wrap around every second and increase a software counter value in its interrupt service routine (triggered with the wrap around event) every time the wrap around occurs.

Note

The implementation of this function is mandatory for the correct execution of the API.

Parameters

hct	A pointer to the high counter value bits representing current time in seconds.
lct	A pointer to the low counter value bits representing current time in microseconds. Range: 0, .., 999999 µsec

9.20.3.2 Timer_SetCallback()

```
status_t Timer_SetCallback (
            timer_cb_t f )
```

Installs an periodic timer callback function.

Installs an periodic timer callback function that is invoked whenever an interval elapses. The callback is the same for any interval, however, the single intervals can be identified by the passed parameter. Passing a zero-pointer removes and disables the callback.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disable the periodic timer callback and thus the automatic starting of measurements from the background.

Parameters

f	The timer callback function.

Returns

Returns the status (STATUS_OK on success).

9.20.3.3 Timer_SetInterval()

```
status_t Timer_SetInterval (
            uint32_t dt_microseconds,
            void * param )
```

Sets the timer interval for a specified callback parameter.

Sets the callback interval for the specified parameter and starts the timer with a new interval. If there is already an interval with the given parameter, the timer is restarted with the given interval. If the same time interval as already set is passed, nothing happens. Passing a interval of 0 disables the timer.
Note that a microsecond granularity for the timer interrupt period is not required. Usually a microseconds granularity is sufficient. The required granularity depends on the targeted frame rate, e.g. in case of more than 1 kHz measurement rate, a granularity of less than a microsecond is required to achieve the given frame rate.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disable the periodic timer callback and thus the automatic starting of measurements from the background.

Parameters

dt_microseconds	The callback interval in microseconds.
param	An abstract parameter to be passed to the callback. This is also the identifier of the given interval.

Returns

Returns the status (STATUS_OK on success).

9.20.3.4 Timer_Start()

```
status_t Timer_Start (
            uint32_t dt_microseconds,
            void * param )
```

Starts the timer for a specified callback parameter.

Sets the callback interval for the specified parameter and starts the timer with a new interval. If there is already an interval with the given parameter, the timer is restarted with the given interval. Passing a interval of 0 disables the timer.

Note that a microsecond granularity for the timer interrupt period is not required. Usually a microseconds granularity is sufficient. The required granularity depends on the targeted frame rate, e.g. in case of more than 1 kHz measurement rate, a granularity of less than a microsecond is required to achieve the given frame rate.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disable the periodic timer callback and thus the automatic starting of measurements from the background.

Parameters

dt_microseconds	The callback interval in microseconds.
param	An abstract parameter to be passed to the callback. This is also the identifier of the given interval.

Returns

Returns the status (STATUS_OK on success).

9.20.3.5 Timer_Stop()

```
status_t Timer_Stop (
            void * param )
```

Stops the timer for a specified callback parameter.

Stops a callback interval for the specified parameter.

Note

The implementation of this function is optional for the correct execution of the API. If not implemented, a weak implementation within the API will be used that disable the periodic timer callback and thus the automatic starting of measurements from the background.

Parameters

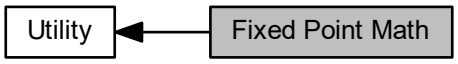
param	An abstract parameter that identifies the interval to be stopped.

Returns

Returns the status (STATUS_OK on success).

9.21 Fixed Point Math

A basic math library for fixed point number in the Qx.y fomat.
Collaboration diagram for Fixed Point Math:



Macros

- #define UQ4_4_MAX ((uq4_4_t)UINT8_MAX)
- #define UQ4_4_ONE ((uq4_4_t)(1U<<4U))
- #define UQ2_6_ONE ((uq2_6_t)(1U<<6U))
- #define UQ0_8_MAX ((uq0_8_t)UINT8_MAX)
- #define UQ12_4_MAX ((uq12_4_t)UINT16_MAX)
- #define UQ12_4_ONE ((uq12_4_t)(1U<<4U))
- #define UQ11_4_ONE ((q11_4_t)(1 << 4))
- #define Q11_4_MAX ((q11_4_t)INT16_MAX)
- #define Q11_4_MIN ((q11_4_t)INT16_MIN)
- #define UQ10_6_MAX ((uq10_6_t)UINT16_MAX)
- #define UQ10_6_ONE ((uq10_6_t)(1U << 6U))
- #define UQ1_15_MAX ((uq1_15_t)UINT16_MAX)
- #define UQ1_15_ONE ((uq1_15_t)(1U << 15U))
- #define UQ0_16_MAX ((uq0_16_t)UINT16_MAX)
- #define UQ0_16_ONE ((uq0_16_t)(1U<<16U))
- #define UQ28_4_MAX ((uq28_4_t)UINT32_MAX)
- #define UQ28_4_ONE ((uq28_4_t)(1U<<4U))
- #define UQ27_4_ONE ((q27_4_t)(1 << 4))
- #define Q27_4_MAX ((q27_4_t)INT32_MAX)
- #define Q27_4_MIN ((q27_4_t)INT32_MIN)
- #define UQ16_16_ONE ((uq16_16_t)(1U << 16U))
- #define UQ16_16_MAX ((uq16_16_t)UINT32_MAX)
- #define UQ16_16_E (0x2B7E1U)
- #define Q15_16_ONE ((q15_16_t)(1 << 16))
- #define Q15_16_MAX ((q15_16_t)INT32_MAX)
- #define Q15_16_MIN ((q15_16_t)INT32_MIN)
- #define Q9_22_ONE ((q9_22_t)(1 << 22))
- #define Q9_22_MAX ((q9_22_t)INT32_MAX)
- #define Q9_22_MIN ((q9_22_t)INT32_MIN)
- #define FP_EXP16_MAX (0x000B1721)
- #define FP_EXP16_MIN (-0x000BC894)
- #define FP_EXP24_MAX (0x058B90BF)
- #define FP_LOG24_2 (0x0B17218)

Typedefs

- typedef uint8_t uq6_2_t

  Unsigned fixed point number: UQ6.2.
- typedef uint8_t uq4_4_t

  Unsigned fixed point number: UQ4.4.
- typedef uint8_t uq2_6_t

  Unsigned fixed point number: UQ2.6.
- typedef uint8_t uq1_7_t

  Unsigned fixed point number: UQ1.7.
- typedef uint8_t uq0_8_t

  Signed fixed point number: Q0.7.
- typedef uint16_t uq12_4_t

  Unsigned fixed point number: UQ12.4.
- typedef int16_t q11_4_t

  Signed fixed point number: Q11.4.
- typedef uint16_t uq10_6_t

  Unsigned fixed point number: UQ10.6.
- typedef uint16_t uq1_15_t

  Unsigned fixed point number: UQ1.15.
- typedef int16_t q0_15_t

  Signed fixed point number: Q0.15.
- typedef int16_t q3_12_t

  Signed fixed point number: Q2.13.
- typedef uint16_t uq0_16_t

  Unsigned fixed point number: UQ0.16.
- typedef uint32_t uq28_4_t

  Unsigned fixed point number: UQ28.4.
- typedef int32_t q27_4_t

  Signed fixed point number: Q27.4.
- typedef uint32_t uq16_16_t

  Unsigned fixed point number: UQ16.16.
- typedef int32_t q15_16_t

  Signed fixed point number: Q15.16.
- typedef uint32_t uq10_22_t

  Unsigned fixed point number: UQ10.22.
- typedef int32_t q9_22_t

  Signed fixed point number: Q9.22.

9.21.1 Detailed Description

A basic math library for fixed point number in the Qx.y fomat.

This module contains common fixed point type definitions as well as some basic math algorithms. All types are based on integer types. The number are defined with the Q number format.

- For a description of the Q number format refer to: https://en.wikipedia.org/wiki/Q_(number_format)
- Another resource for fixed point math in C might be found at http://www.eetimes.com/author.asp?section_id=36&doc_id=1287491

Warning

This definitions are not portable and work only with little-endian systems!

9.21.2 Macro Definition Documentation

9.21.2.1 FP_EXP16_MAX

```
#define FP_EXP16_MAX (0x000B1721)
```

The maximum number for the input of the fp_exp16 function that gives a result still larger than 0 (in UQ16.16 representation).

9.21.2.2 FP_EXP16_MIN

```
#define FP_EXP16_MIN (-0x000BC894)
```

The minimum number for the input of the fp_exp16 function that gives a result smaller than the maximum of the UQ16.16 format.

9.21.2.3 FP_EXP24_MAX

```
#define FP_EXP24_MAX (0x058B90BF)
```

The maximum number for the input of the fp_exp24 function that gives a result still larger than 0 (in UQ08.24 representation).

9.21.2.4 FP_LOG24_2

```
#define FP_LOG24_2 (0x0B17218)
```

The natural logarithm of 2 in UQ08.24 representation.

9.21.2.5 Q11_4_MAX

```
#define Q11_4_MAX ((q11_4_t)INT16_MAX)
```

Maximum value of Q11.4 number format.

9.21.2.6 Q11_4_MIN

```
#define Q11_4_MIN ((q11_4_t)INT16_MIN)
```

Minimum value of Q11.4 number format.

9.21.2.7 Q15_16_MAX

```
#define Q15_16_MAX ((q15_16_t)INT32_MAX)
```

Maximum value of Q15.16 number format.

9.21.2.8 Q15_16_MIN

```
#define Q15_16_MIN ((q15_16_t)INT32_MIN)
```

Minimum value of Q15.16 number format.

9.21.2.9 Q15_16_ONE

```
#define Q15_16_ONE ((q15_16_t)(1 << 16))
```

The 1/one/unity in Q15.16 number format.

9.21.2.10 Q27_4_MAX

```
#define Q27_4_MAX ((q27_4_t)INT32_MAX)
```

Maximum value of Q27.4 number format.

9.21.2.11 Q27_4_MIN

```
#define Q27_4_MIN ((q27_4_t)INT32_MIN)
```
Minimum value of Q27.4 number format.

9.21.2.12 Q9_22_MAX

```
#define Q9_22_MAX ((q9_22_t)INT32_MAX)
```
Maximum value of Q9.22 number format.

9.21.2.13 Q9_22_MIN

```
#define Q9_22_MIN ((q9_22_t)INT32_MIN)
```
Minimum value of Q9.22 number format.

9.21.2.14 Q9_22_ONE

```
#define Q9_22_ONE ((q9_22_t)(1 << 22))
```
The 1/one/unity in Q9.22 number format.

Examples

01_simple_example.c, and 02_advanced_example.c.

9.21.2.15 UQ0_16_MAX

```
#define UQ0_16_MAX ((uq0_16_t)UINT16_MAX)
```
Maximum value of UQ0.16 number format.

9.21.2.16 UQ0_16_ONE

```
#define UQ0_16_ONE ((uq0_16_t)(1U<<16U))
```
The 1/one/unity in UQ0.16 number format.

9.21.2.17 UQ0_8_MAX

```
#define UQ0_8_MAX ((uq0_8_t)UINT8_MAX)
```
Maximum value of UQ0.8 number format.

9.21.2.18 UQ10_6_MAX

```
#define UQ10_6_MAX ((uq10_6_t)UINT16_MAX)
```
Maximum value of UQ10.6 number format.

9.21.2.19 UQ10_6_ONE

```
#define UQ10_6_ONE ((uq10_6_t)(1U << 6U))
```
The 1/one/unity in UQ10.6 number format.

9.21.2.20 UQ11_4_ONE

```
#define UQ11_4_ONE ((q11_4_t)(1 << 4))
```
The 1/one/unity in UQ11.4 number format.

9.21.2.21 UQ12_4_MAX

```
#define UQ12_4_MAX ((uq12_4_t)UINT16_MAX)
```
Maximum value of UQ12.4 number format.

9.21.2.22 UQ12_4_ONE

```
#define UQ12_4_ONE ((uq12_4_t)(1U<<4U))
```
The 1/one/unity in UQ12.4 number format.

9.21.2.23 UQ16_16_E

```
#define UQ16_16_E (0x2B7E1U)
```
Euler's number, e, in UQ16.16 format.

9.21.2.24 UQ16_16_MAX

```
#define UQ16_16_MAX ((uq16_16_t)UINT32_MAX)
```
Maximum value of UQ16.16 number format.

9.21.2.25 UQ16_16_ONE

```
#define UQ16_16_ONE ((uq16_16_t)(1U << 16U))
```
The 1/one/unity in UQ16.16 number format.

9.21.2.26 UQ1_15_MAX

```
#define UQ1_15_MAX ((uq1_15_t)UINT16_MAX)
```
Maximum value of UQ1.15 number format.

9.21.2.27 UQ1_15_ONE

```
#define UQ1_15_ONE ((uq1_15_t)(1U << 15U))
```
The 1/one/unity in UQ1.15 number format.

9.21.2.28 UQ27_4_ONE

```
#define UQ27_4_ONE ((q27_4_t)(1 << 4))
```
The 1/one/unity in UQ27.4 number format.

9.21.2.29 UQ28_4_MAX

```
#define UQ28_4_MAX ((uq28_4_t)UINT32_MAX)
```
Maximum value of UQ28.4 number format.

9.21.2.30 UQ28_4_ONE

```
#define UQ28_4_ONE ((uq28_4_t)(1U<<4U))
```
The 1/one/unity in UQ28.4 number format.

9.21.2.31 UQ2_6_ONE

```
#define UQ2_6_ONE ((uq2_6_t)(1U<<6U))
```
The 1/one/unity in UQ2.6 number format.

9.21.2.32 UQ4_4_MAX

```
#define UQ4_4_MAX ((uq4_4_t)UINT8_MAX)
```
Maximum value of UQ4.4 number format.

9.21.2.33 UQ4_4_ONE

```
#define UQ4_4_ONE ((uq4_4_t)(1U<<4U))
```
The 1/one/unity in UQ4.4 number format.

9.21.3 Typedef Documentation

9.21.3.1 q0_15_t

```
typedef int16_t q0_15_t
```

Signed fixed point number: Q0.15.
An signed fixed point number format based on the 16-bit integer type with 0 integer and 15 fractional bits.

- Range: -1 .. 0.999969482
- Granularity: 0.000030518

9.21.3.2 q11_4_t

```
typedef int16_t q11_4_t
```

Signed fixed point number: Q11.4.
An signed fixed point number format based on the 16-bit signed integer type with 11 integer and 4 fractional bits.

- Range: -2048 ... 2047.9375
- Granularity: 0.0625

9.21.3.3 q15_16_t

```
typedef int32_t q15_16_t
```

Signed fixed point number: Q15.16.
An signed fixed point number format based on the 32-bit integer type with 15 integer and 16 fractional bits.

- Range: -32768 .. 32767.99998
- Granularity: 1.52588E-05

9.21.3.4 q27_4_t

```
typedef int32_t q27_4_t
```

Signed fixed point number: Q27.4.
An signed fixed point number format based on the 32-bit signed integer type with 27 integer and 4 fractional bits.

- Range: -134217728 ... 134217727.9375
- Granularity: 0.0625

9.21.3.5 q3_12_t

```
typedef int16_t q3_12_t
```

Signed fixed point number: Q2.13.
An signed fixed point number format based on the 16-bit integer type with 2 integer and 13 fractional bits.

- Range: -4 .. 3.99987793
- Granularity: 0.00012207

Signed fixed point number: Q13.2
An signed fixed point number format based on the 16-bit integer type with 13 integer and 2 fractional bits.

- Range: -8192 .. 8191.75

- Granularity: 0.25

Signed fixed point number: Q3.12
An signed fixed point number format based on the 16-bit integer type with 3 integer and 12 fractional bits.

- Range: -8 .. 7.99975586
- Granularity: 0.00024414

9.21.3.6 q9_22_t

```
typedef int32_t q9_22_t
```

Signed fixed point number: Q9.22.
An signed fixed point number format based on the 32-bit integer type with 9 integer and 22 fractional bits.

- Range: -512 ... 511.9999998
- Granularity: 2.38418579101562E-07

9.21.3.7 uq0_16_t

```
typedef uint16_t uq0_16_t
```

Unsigned fixed point number: UQ0.16.
An unsigned fixed point number format based on the 16-bit unsigned integer type with 0 integer and 16 fractional bits.

- Range: 0 .. 0.9999847412109375
- Granularity: 1.52587890625e-5

9.21.3.8 uq0_8_t

```
typedef uint8_t uq0_8_t
```

Signed fixed point number: Q0.7.
An signed fixed point number format based on the 8-bit integer type with 0 integer and 7 fractional bits.

- Range: -1 .. 0.9921875
- Granularity: 0.0078125

Unsigned fixed point number: UQ0.8
An unsigned fixed point number format based on the 8-bit unsigned integer type with 1 integer and 7 fractional bits.

- Range: 0 .. 0.99609375
- Granularity: 0.00390625

9.21.3.9 uq10_22_t

```
typedef uint32_t uq10_22_t
```

Unsigned fixed point number: UQ10.22.
An unsigned fixed point number format based on the 32-bit unsigned integer type with 10 integer and 22 fractional bits.

- Range: 0 ... 1023.99999976158
- Granularity: 2.38418579101562E-07

9.21.3.10 uq10_6_t

```
typedef uint16_t uq10_6_t
```

Unsigned fixed point number: UQ10.6.
An unsigned fixed point number format based on the 16-bit unsigned integer type with 10 integer and 6 fractional bits.

- Range: 0 ... 1023.984375
- Granularity: 0.015625

9.21.3.11 uq12_4_t

```
typedef uint16_t uq12_4_t
```

Unsigned fixed point number: UQ12.4.
An unsigned fixed point number format based on the 16-bit unsigned integer type with 12 integer and 4 fractional bits.

- Range: 0 ... 4095.9375
- Granularity: 0.0625

9.21.3.12 uq16_16_t

```
typedef uint32_t uq16_16_t
```

Unsigned fixed point number: UQ16.16.
An unsigned fixed point number format based on the 32-bit unsigned integer type with 16 integer and 16 fractional bits.

- Range: 0 ... 65535.999984741
- Granularity: 0.000015259

9.21.3.13 uq1_15_t

```
typedef uint16_t uq1_15_t
```

Unsigned fixed point number: UQ1.15.
An unsigned fixed point number format based on the 16-bit unsigned integer type with 1 integer and 15 fractional bits.

- Range: 0 .. 1.999969
- Granularity: 0.000031

9.21.3.14 uq1_7_t

```
typedef uint8_t uq1_7_t
```

Unsigned fixed point number: UQ1.7.
An unsigned fixed point number format based on the 8-bit unsigned integer type with 1 integer and 7 fractional bits.

- Range: 0 .. 1.9921875
- Granularity: 0.0078125

9.21.3.15 uq28_4_t

```
typedef uint32_t uq28_4_t
```

Unsigned fixed point number: UQ28.4.
An unsigned fixed point number format based on the 32-bit unsigned integer type with 28 integer and 4 fractional bits.

- Range: 0 ... 268435455.9375
- Granularity: 0.0625

9.21.3.16 uq2_6_t

```
typedef uint8_t uq2_6_t
```

Unsigned fixed point number: UQ2.6.
An unsigned fixed point number format based on the 8-bit unsigned integer type with 2 integer and 6 fractional bits.

- Range: 0 .. 3.984375
- Granularity: 0.015625

9.21.3.17 uq4_4_t

```
typedef uint8_t uq4_4_t
```

Unsigned fixed point number: UQ4.4.
An unsigned fixed point number format based on the 8-bit unsigned integer type with 4 integer and 4 fractional bits.

- Range: 0 .. 15.9375
- Granularity: 0.0625

9.21.3.18 uq6_2_t

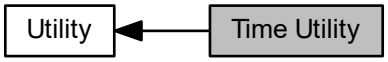
```
typedef uint8_t uq6_2_t
```

Unsigned fixed point number: UQ6.2.
An unsigned fixed point number format based on the 8-bit unsigned integer type with 6 integer and 2 fractional bits.

- Range: 0 .. 63.75
- Granularity: 0.25

9.22 Time Utility

Timer utilities for time measurement duties.
Collaboration diagram for Time Utility:



Data Structures

- struct ltc_t

  A data structure to represent current time.

Functions

- void Time_GetNow (ltc_t *t_now)

  Obtains the elapsed time since MCU startup.
- uint32_t Time_GetNowUSec (void)

  Obtains the elapsed microseconds since MCU startup.
- uint32_t Time_GetNowMSec (void)

  Obtains the elapsed milliseconds since MCU startup.
- uint32_t Time_GetNowSec (void)

  Obtains the elapsed seconds since MCU startup.
- void Time_GetElapsed (ltc_t *t_elapsed, ltc_t *const t_start)

  Obtains the elapsed time since a given time point.
- uint32_t Time_GetElapsedUSec (ltc_t *const t_start)

  Obtains the elapsed microseconds since a given time point.
- uint32_t Time_GetElapsedMSec (ltc_t *const t_start)

  Obtains the elapsed milliseconds since a given time point.
- uint32_t Time_GetElapsedSec (ltc_t *const t_start)

  Obtains the elapsed seconds since a given time point.
- void Time_Diff (ltc_t *t_diff, ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points.
- uint32_t Time_DiffUSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in microseconds.
- uint32_t Time_DiffMSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in milliseconds.
- uint32_t Time_DiffSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in seconds.
- void Time_Delay (ltc_t const *dt)

  Time delay for a given time period.
- void Time_DelayUSec (uint32_t dt_usec)

  Time delay for a given time period in microseconds.
- void Time_DelayMSec (uint32_t dt_msec)

  Time delay for a given time period in milliseconds.
- void Time_DelaySec (uint32_t dt_sec)

Time delay for a given time period in seconds.

- bool Time_CheckTimeout (ltc_t const *t_start, ltc_t const *t_timeout)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutUSec (ltc_t const *t_start, uint32_t const t_timeout_usec)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutMSec (ltc_t const *t_start, uint32_t const t_timeout_msec)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutSec (ltc_t const *t_start, uint32_t const t_timeout_sec)

  Checks if timeout is reached from a given starting time.
- void Time_Add (ltc_t *t, ltc_t const *t1, ltc_t const *t2)

  Adds two ltc_t values.
- void Time_AddUSec (ltc_t *t, ltc_t const *t1, uint32_t t2_usec)

  Adds a given time in microseconds to an ltc_t value.
- void Time_AddMSec (ltc_t *t, ltc_t const *t1, uint32_t t2_msec)

  Adds a given time in milliseconds to an ltc_t value.
- void Time_AddSec (ltc_t *t, ltc_t const *t1, uint32_t t2_sec)

  Adds a given time in seconds to an ltc_t value.
- uint32_t Time_ToUSec (ltc_t const *t)

  Converts ltc_t to microseconds (uint32_t).
- uint32_t Time_ToMSec (ltc_t const *t)

  Converts ltc_t to milliseconds (uint32_t).
- uint32_t Time_ToSec (ltc_t const *t)

  Converts ltc_t to seconds (uint32_t).

9.22.1 Detailed Description

Timer utilities for time measurement duties.

This module provides time measurement utility functions like delay or time measurement methods, or time math functions.

9.22.2 Function Documentation

9.22.2.1 Time_Add()

```
void Time_Add (
            ltc_t * t,
            ltc_t const * t1,
            ltc_t const * t2 )
```

Adds two ltc_t values.

Result is defined as t = t1 + t2. Results are wrapped around at maximum values just like integers.

Parameters

t	Return value: t = t1 + t2.
t1	1st operand.
t2	2nd operand.

9.22.2.2 Time_AddMSec()

```
void Time_AddMSec (
            ltc_t * t,
```

```
            ltc_t const * t1,
            uint32_t t2_msec )
```

Adds a given time in milliseconds to an ltc_t value.

Parameters

t	Return value: t = t1 + t2.
t1	1st operand.
t2_msec	2nd operand in milliseconds.

9.22.2.3 Time_AddSec()

```
void Time_AddSec (
            ltc_t * t,
            ltc_t const * t1,
            uint32_t t2_sec )
```

Adds a given time in seconds to an ltc_t value.

Parameters

t	Return value: t = t1 + t2.
t1	1st operand.
t2_sec	2nd operand in seconds.

9.22.2.4 Time_AddUSec()

```
void Time_AddUSec (
            ltc_t * t,
            ltc_t const * t1,
            uint32_t t2_usec )
```

Adds a given time in microseconds to an ltc_t value.

Parameters

t	Return value: t = t1 + t2.
t1	1st operand.
t2_usec	2nd operand in microseconds.

9.22.2.5 Time_CheckTimeout()

```
bool Time_CheckTimeout (
            ltc_t const * t_start,
            ltc_t const * t_timeout )
```

Checks if timeout is reached from a given starting time.
Handles overflow.

Parameters

t_start	Start time.
t_timeout	Timeout period.

Returns

Timeout elapsed? True/False (boolean value)

9.22.2.6 Time_CheckTimeoutMSec()

```
bool Time_CheckTimeoutMSec (
            ltc_t const * t_start,
            uint32_t const t_timeout_msec )
```

Checks if timeout is reached from a given starting time.
Handles overflow.

Parameters

t_start	Start time.
t_timeout_msec	Timeout period in milliseconds.

Returns

Timeout elapsed? True/False (boolean value)

9.22.2.7 Time_CheckTimeoutSec()

```
bool Time_CheckTimeoutSec (
            ltc_t const * t_start,
            uint32_t const t_timeout_sec )
```

Checks if timeout is reached from a given starting time.
Handles overflow.

Parameters

t_start	Start time.
t_timeout_sec	Timeout period in seconds.

Returns

Timeout elapsed? True/False (boolean value)

9.22.2.8 Time_CheckTimeoutUSec()

```
bool Time_CheckTimeoutUSec (
            ltc_t const * t_start,
            uint32_t const t_timeout_usec )
```

Checks if timeout is reached from a given starting time.
Handles overflow.

Parameters

t_start	Start time.
t_timeout_usec	Timeout period in microseconds.

Returns

Timeout elapsed? True/False (boolean value)

9.22.2.9 Time_Delay()

```
void Time_Delay (
            ltc_t const * dt )
```

Time delay for a given time period.

Parameters

dt	Delay time.

9.22.2.10 Time_DelayMSec()

```
void Time_DelayMSec (
            uint32_t dt_msec )
```

Time delay for a given time period in milliseconds.

Parameters

dt_msec	Delay time in milliseconds.

9.22.2.11 Time_DelaySec()

```
void Time_DelaySec (
            uint32_t dt_sec )
```

Time delay for a given time period in seconds.

Parameters

dt_sec	Delay time in seconds.

9.22.2.12 Time_DelayUSec()

```
void Time_DelayUSec (
            uint32_t dt_usec )
```

Time delay for a given time period in microseconds.

Parameters

dt_usec	Delay time in microseconds.

9.22.2.13 Time_Diff()

```
void Time_Diff (
            ltc_t * t_diff,
```

```
            ltc_t const * t_start,
            ltc_t const * t_end )
```

Obtains the time difference between two given time points.
Result is defined as t_diff = t_end - t_start. Note: since no negative time differences are supported, t_end has to be later/larger than t_start. Otherwise, the result won't be a negative time span but given by max value.

Parameters

t_diff	Returned time difference.
t_start	Start time point.
t_end	End time point.

9.22.2.14 Time_DiffMSec()

```
uint32_t Time_DiffMSec (
            ltc_t const * t_start,
            ltc_t const * t_end )
```

Obtains the time difference between two given time points in milliseconds.
Result is defined as t_diff = t_end - t_start. Refers to Time_Diff() and handles overflow. Wrap around effect due to uint32_t result format!!

Parameters

t_start	Start time point.
t_end	End time point.

Returns

Time difference in milliseconds.

9.22.2.15 Time_DiffSec()

```
uint32_t Time_DiffSec (
            ltc_t const * t_start,
            ltc_t const * t_end )
```

Obtains the time difference between two given time points in seconds.
Result is defined as t_diff = t_end - t_start. Refers to Time_Diff() and handles overflow.

Parameters

t_start	Start time point.
t_end	End time point.

Returns

Time difference in seconds.

9.22.2.16 Time_DiffUSec()

```
uint32_t Time_DiffUSec (
            ltc_t const * t_start,
            ltc_t const * t_end )
```

Obtains the time difference between two given time points in microseconds.
Result is defined as t_diff = t_end - t_start. Refers to Time_Diff() and handles overflow such that to large values are limited by 0xFFFFFFFF µs.

Parameters

t_start	Start time point.
t_end	End time point.

Returns

Time difference in microseconds.

9.22.2.17 Time_GetElapsed()

```
void Time_GetElapsed (
            ltc_t * t_elapsed,
            ltc_t *const t_start )
```

Obtains the elapsed time since a given time point.

Parameters

t_elapsed	Returns the elapsed time since t_start.
t_start	Start time point.

9.22.2.18 Time_GetElapsedMSec()

```
uint32_t Time_GetElapsedMSec (
            ltc_t *const t_start )
```

Obtains the elapsed milliseconds since a given time point.
Wrap around effect due to uint32_t result format!!

Parameters

t_start	Start time point.

Returns

Elapsed milliseconds since t_start as uint32_t.

9.22.2.19 Time_GetElapsedSec()

```
uint32_t Time_GetElapsedSec (
            ltc_t *const t_start )
```

Obtains the elapsed seconds since a given time point.

Parameters

t_start	Start time point.

Returns

Elapsed seconds since t_start as uint32_t.

9.22.2.20 Time_GetElapsedUSec()

```
uint32_t Time_GetElapsedUSec (
            ltc_t *const t_start )
```

Obtains the elapsed microseconds since a given time point.
Wrap around effect due to uint32_t result format!!

Parameters

t_start	Start time point.

Returns

Elapsed microseconds since t_start as uint32_t.

9.22.2.21 Time_GetNow()

```
void Time_GetNow (
            ltc_t * t_now )
```

Obtains the elapsed time since MCU startup.

Parameters

t_now	returned current time

9.22.2.22 Time_GetNowMSec()

```
uint32_t Time_GetNowMSec (
            void  )
```

Obtains the elapsed milliseconds since MCU startup.
Wrap around effect due to uint32_t result format!!

Parameters

-	

Returns

Elapsed milliseconds since MCU startup as uint32_t.

9.22.2.23 Time_GetNowSec()

```
uint32_t Time_GetNowSec (
            void  )
```

Obtains the elapsed seconds since MCU startup.

Parameters

-	

Returns

Elapsed seconds since MCU startup as uint32_t.

9.22.2.24 Time_GetNowUSec()

```
uint32_t Time_GetNowUSec (
            void  )
```

Obtains the elapsed microseconds since MCU startup.
Wrap around effect due to uint32_t result format!!

Parameters

-	

Returns

Elapsed microseconds since MCU startup as uint32_t.

9.22.2.25 Time_ToMSec()

```
uint32_t Time_ToMSec (
            ltc_t const * t )
```

Converts ltc_t to milliseconds (uint32_t).

Parameters

t	Input ltc_t struct.

Returns

Time value in milliseconds.

9.22.2.26 Time_ToSec()

```
uint32_t Time_ToSec (
            ltc_t const * t )
```

Converts ltc_t to seconds (uint32_t).

Parameters

t	Input ltc_t struct.

Returns

Time value in seconds.

9.22.2.27 Time_ToUSec()

```
uint32_t Time_ToUSec (
            ltc_t const * t )
```

Converts ltc_t to microseconds (uint32_t).

Parameters

t	Input ltc_t struct.

Returns

Time value in microseconds.

Chapter 10

Data Structure Documentation

10.1 argus_cal_p2pxtalk_t Struct Reference

Pixel-To-Pixel Crosstalk Compensation Parameters.
`#include <argus_xtalk.h>`

Data Fields

- bool Enabled
- uq0_8_t RelativeThreshold
- uq12_4_t AbsoluteTreshold
- q3_12_t KcFactorS
- q3_12_t KcFactorC
- q3_12_t KcFactorSRefPx
- q3_12_t KcFactorCRefPx

10.1.1 Detailed Description

Pixel-To-Pixel Crosstalk Compensation Parameters.
Contains calibration data that belongs to the pixel-to-pixel crosstalk compensation feature.

10.1.2 Field Documentation

10.1.2.1 AbsoluteTreshold

`uq12_4_t argus_cal_p2pxtalk_t::AbsoluteTreshold`
The absolute threshold determines the minimum total crosstalk amplitude (i.e. the average amplitude of all pixels weighted by the Kc factor) that is required for the compensation to become active. Set to 0 to always enable. Absolute and relative conditions are connected with AND logic.

10.1.2.2 Enabled

`bool argus_cal_p2pxtalk_t::Enabled`
Pixel-To-Pixel Compensation on/off.

10.1.2.3 KcFactorC

`q3_12_t argus_cal_p2pxtalk_t::KcFactorC`
The cosine component of the Kc factor that determines the amount of the total signal of all pixels that influences the individual signal of each pixel. Higher values determine more influence on the individual pixel signal.

10.1.2.4 KcFactorCRefPx

q3_12_t argus_cal_p2pxtalk_t::KcFactorCRefPx
The cosine component of the reference pixel Kc factor that determines the amount of the total signal on all pixels that influences the individual signal of the reference pixel. Higher values determine more influence on the reference pixel signal.

10.1.2.5 KcFactorS

q3_12_t argus_cal_p2pxtalk_t::KcFactorS
The sine component of the Kc factor that determines the amount of the total signal of all pixels that influences the individual signal of each pixel. Higher values determine more influence on the individual pixel signal.

10.1.2.6 KcFactorSRefPx

q3_12_t argus_cal_p2pxtalk_t::KcFactorSRefPx
The sine component of the reference pixel Kc factor that determines the amount of the total signal on all pixels that influences the individual signal of the reference pixel. Higher values determine more influence on the reference pixel signal.

10.1.2.7 RelativeThreshold

uq0_8_t argus_cal_p2pxtalk_t::RelativeThreshold
The relative threshold determines when the compensation is active for each individual pixel. The value determines the ratio of the individual pixel signal is with respect to the overall average signal. If the ratio is smaller than the value, the compensation is active. Absolute and relative conditions are connected with AND logic.
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_xtalk.h

10.2 argus_cfg_dca_t Struct Reference

Dynamic Configuration Adaption (DCA) Parameters.
`#include <argus_dca.h>`

Data Fields

- argus_dca_enable_t Enabled
- uint8_t SatPxThLin
- uint8_t SatPxThExp
- uint8_t SatPxThRst
- uq12_4_t Atarget
- uq12_4_t AthLow
- uq12_4_t AthHigh
- uq10_6_t DepthNom
- uq10_6_t DepthMin
- uq10_6_t DepthMax
- uq12_4_t PowerNom
- uq12_4_t PowerMin
- argus_dca_gain_t GainNom
- argus_dca_gain_t GainMin
- argus_dca_gain_t GainMax
- uq0_8_t PowerSavingRatio

10.2.1 Detailed Description

Dynamic Configuration Adaption (DCA) Parameters.
DCA contains:

- Static or dynamic mode. The first is utilizing the nominal values while the latter is dynamically adopting between min. and max. value and starting form the nominal values.
- Analog Integration Depth Adaption down to single pulses.
- Optical Output Power Adaption
- Pixel Input Gain Adaption
- Digital Integration Depth Adaption
- Dynamic Global Phase Shift Injection.
- All that features are heeding the Laser Safety limits.

10.2.2 Field Documentation

10.2.2.1 Atarget

`uq12_4_t argus_cfg_dca_t::Atarget`
The amplitude to be targeted from the lower regime. If the amplitude lower than the target value, a linear increase of integration energy will happen in order to optimize for best performance.
Valid values: ARGUS_CFG_DCA_ATH_MIN, ... ARGUS_CFG_DCA_ATH_MAX or 0 Set 0 to disable optimization toward the target amplitude. Note further that the following condition must hold: 'MIN' <= AthLow <= Atarget <= AthHigh <= 'MAX'

10.2.2.2 AthHigh

`uq12_4_t argus_cfg_dca_t::AthHigh`
The high threshold value for the max. amplitude. If the max. amplitude exceeds this value, the integration depth will be decreases. Note that also saturated pixels will cause a decrease of the integration depth.
Valid values: ARGUS_CFG_DCA_ATH_MIN, ... ARGUS_CFG_DCA_ATH_MAX Note further that the following condition must hold: 'MIN' <= AthLow <= Atarget <= AthHigh <= 'MAX'

10.2.2.3 AthLow

`uq12_4_t argus_cfg_dca_t::AthLow`
The low threshold value for the max. amplitude. If the max. amplitude falls below this value, the integration depth will be increases.
Valid values: ARGUS_CFG_DCA_ATH_MIN, ... ARGUS_CFG_DCA_ATH_MAX Note further that the following condition must hold: 'MIN' <= AthLow <= Atarget <= AthHigh <= 'MAX'

10.2.2.4 DepthMax

`uq10_6_t argus_cfg_dca_t::DepthMax`
The maximum analog integration depth in UQ10.6 format, i.e. the maximum pattern count per sample.
Valid values: ARGUS_CFG_DCA_DEPTH_MIN, ... ARGUS_CFG_DCA_DEPTH_MAX Note further that the following condition must hold: 'MIN' <= DepthMin <= DepthNom <= DepthMax <= 'MAX'

10.2.2.5 DepthMin

`uq10_6_t argus_cfg_dca_t::DepthMin`
The minimum analog integration depth in UQ10.6 format, i.e. the minimum pattern count per sample.
Valid values: ARGUS_CFG_DCA_DEPTH_MIN, ... ARGUS_CFG_DCA_DEPTH_MAX Note further that the following condition must hold: 'MIN' <= DepthLow <= DepthNom <= DepthHigh <= 'MAX'

10.2.2.6 DepthNom

uq10_6_t argus_cfg_dca_t::DepthNom
The nominal analog integration depth in UQ10.6 format, i.e. the nominal pattern count per sample.
Valid values: ARGUS_CFG_DCA_DEPTH_MIN, ... ARGUS_CFG_DCA_DEPTH_MAX Note further that the following condition must hold: 'MIN' <= DepthLow <= DepthNom <= DepthHigh <= 'MAX'

10.2.2.7 Enabled

argus_dca_enable_t argus_cfg_dca_t::Enabled
Enables the automatic configuration adaption features. Enables the dynamic part if DCA_ENABLE_DYNAMIC and the static only if DCA_ENABLE_STATIC. If set to DCA_ENABLE_OFF, the DCA is completely skipped and the static register values are considered which is recommended for advanced debugging only.

10.2.2.8 GainMax

argus_dca_gain_t argus_cfg_dca_t::GainMax
The maximum pixel gain setting, i.e. the setting for maximum gain stage.
Valid values: 0,..,3: DCA_GAIN_LOW, ... DCA_GAIN_HIGH Note further that the following condition must hold: 'MIN' <= GainMin <= GainNom <= GainMax <= 'MAX'

10.2.2.9 GainMin

argus_dca_gain_t argus_cfg_dca_t::GainMin
The minimal pixel gain setting, i.e. the setting for minimum gain stage.
Valid values: 0,..,3: DCA_GAIN_LOW, ... DCA_GAIN_HIGH Note further that the following condition must hold: 'MIN' <= GainMin <= GainNom <= GainMax <= 'MAX'

10.2.2.10 GainNom

argus_dca_gain_t argus_cfg_dca_t::GainNom
The nominal pixel gain setting, i.e. the setting for nominal/default gain stage.
Valid values: 0,..,3: DCA_GAIN_LOW, ... DCA_GAIN_HIGH Note further that the following condition must hold: 'MIN' <= GainMin <= GainNom <= GainMax <= 'MAX'

10.2.2.11 PowerMin

uq12_4_t argus_cfg_dca_t::PowerMin
The minimum optical output power in mA, i.e. the minimum VCSEL_HC1 setting.
Valid values: ARGUS_CFG_DCA_POWER_MIN, ... ARGUS_CFG_DCA_POWER_MAX Note further that the following condition must hold: 'MIN' <= PowerMin <= PowerNom <= 'MAX'

10.2.2.12 PowerNom

uq12_4_t argus_cfg_dca_t::PowerNom
The nominal optical output power in mA, i.e. the nominal VCSEL_HC1 setting.
Valid values: ARGUS_CFG_DCA_POWER_MIN, ... ARGUS_CFG_DCA_POWER_MAX Note further that the following condition must hold: 'MIN' <= PowerMin <= PowerNom <= 'MAX'

10.2.2.13 PowerSavingRatio

uq0_8_t argus_cfg_dca_t::PowerSavingRatio
Power Saving Ratio value.
Determines the percentage of the full available frame time that is not exploited for digital integration. Thus the device is idle within the specified portion of the frame time and does consume less energy.
Note that the laser safety might already limit the maximum integration depth and the power saving ratio might not take effect for all ambient situations. Thus the Power Saving Ratio is to be understood as a minimum percentage where the device is idle per frame.

The value is a UQ0.8 format that ranges from 0.0 (=0x00) to 0.996 (=0xFF), where 0 means no power saving (i.e. feature disabled) and 0xFF determines maximum power saving, i.e. the digital integration depth is limited to a single sample.
Range: 0x00, .., 0xFF; set 0 to disable.

10.2.2.14 SatPxThExp

`uint8_t argus_cfg_dca_t::SatPxThExp`
The threshold number of saturated pixels that causes a exponential reduction of the integration energy, i.e. if the number of saturated pixels is larger or equal to this value, the integration energy will be halved.
Valid values: 1, ..., 33; (use 33 to disable the exponential decrease) Note that the exponential value must be between the linear and reset values. To sum up, it must hold: 1 <= SatPxThLin <= SatPxThExp <= SatPxThRst <= 33

10.2.2.15 SatPxThLin

`uint8_t argus_cfg_dca_t::SatPxThLin`
The threshold value of saturated pixels that causes a linear reduction of the integration energy, i.e. if the number of saturated pixels are larger or equal to this value, the integration energy will be reduced by a single step (one pattern if the current integration depth is > 1, one pulse if the current integration depth is <= 1 or one power LSB for the optical power range).
Valid values: 1, ..., 33; (use 33 to disable the linear decrease) Note that the linear value must be smaller or equal to the exponential value. To sum up, it must hold: 1 <= SatPxThLin <= SatPxThExp <= SatPxThRst <= 33

10.2.2.16 SatPxThRst

`uint8_t argus_cfg_dca_t::SatPxThRst`
The threshold number of saturated pixels that causes a sudden reset of the integration energy to the minimal value, i.e. if the number of saturated pixels are larger or equal to this value, the integration energy will suddenly be reset to the minimum values. The gain setting will stay at the mid value and a decrease happens after the next step if still required.
Valid values: 1, ..., 33; (use 33 to disable the sudden reset) Note that the reset value must be larger or equal to the exponential value. To sum up, it must hold: 1 <= SatPxThLin <= SatPxThExp <= SatPxThRst <= 33
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_dca.h

10.3 argus_cfg_pba_t Struct Reference

The pixel binning algorithm settings data structure.
`#include <argus_pba.h>`

Data Fields

- argus_pba_flags_t Enabled
- argus_pba_averaging_mode_t Mode
- uq0_8_t RelAmplThreshold
- uq0_8_t RelMinDistanceScope
- uq12_4_t AbsAmplThreshold
- uq1_15_t AbsMinDistanceScope
- uint32_t PrefilterMask

10.3.1 Detailed Description

The pixel binning algorithm settings data structure.
Describes the pixel binning algorithm settings.

10.3.2 Field Documentation

10.3.2.1 AbsAmplThreshold

uq12_4_t argus_cfg_pba_t::AbsAmplThreshold
The Absolute amplitude threshold value in LSB. Pixels with amplitude below this threshold value are dismissed.
All available values from the 16-bit representation are valid. The actual LSB value is determined by x/16.
Use 0 to disable the absolute amplitude threshold.

10.3.2.2 AbsMinDistanceScope

uq1_15_t argus_cfg_pba_t::AbsMinDistanceScope
The absolute minimum distance scope value in m. Pixels that have a range value within [x0, x0 + dx] are considered for the pixel binning, where x0 is the minimum distance of all amplitude picked pixels and dx is the minimum distance scope value. The minimum distance scope value will be the maximum of relative and absolute value.
All available values from the 16-bit representation are valid. The actual LSB value is determined by x/2^15.
Special values:

- 0: Use 0 for relative value only or to choose the pixel with the minimum distance only (of also the relative value is 0)!

10.3.2.3 Enabled

argus_pba_flags_t argus_cfg_pba_t::Enabled
Enables the pixel binning features. Each bit may enable a different feature. See argus_pba_flags_t for details about the enabled flags.

10.3.2.4 Mode

argus_pba_averaging_mode_t argus_cfg_pba_t::Mode
Determines the PBA averaging mode which is used to obtain the final range value from the algorithm, for example, the average of all pixels. See argus_pba_averaging_mode_t for more details about the individual evaluation modes.

10.3.2.5 PrefilterMask

uint32_t argus_cfg_pba_t::PrefilterMask
The pre-filter pixel mask determines the pixel channels that are statically excluded from the pixel binning (i.e. 1D distance) result.
The pixel enabled mask is an 32-bit mask that determines the device internal channel number. It is recommended to use the

- PIXELXY_ISENABLED(msk, x, y)
- PIXELXY_ENABLE(msk, x, y)
- PIXELXY_DISABLE(msk, x, y)

macros to work with the pixel enable masks.

10.3.2.6 RelAmplThreshold

uq0_8_t argus_cfg_pba_t::RelAmplThreshold
The Relative amplitude threshold value (in %) of the max. amplitude. Pixels with amplitude below this threshold value are dismissed.
All available values from the 8-bit representation are valid. The actual percentage value is determined by 100%/256∗x.
Use 0 to disable the relative amplitude threshold.

10.3.2.7 RelMinDistanceScope

`uq0_8_t argus_cfg_pba_t::RelMinDistanceScope`

The relative minimum distance scope value in %. Pixels that have a range value within [x0, x0 + dx] are considered for the pixel binning, where x0 is the minimum distance of all amplitude picked pixels and dx is the minimum distance scope value. The minimum distance scope value will be the maximum of relative and absolute value.
All available values from the 8-bit representation are valid. The actual percentage value is determined by 100%/256∗x.
Special values:

- 0: Use 0 for absolute value only or to choose the pixel with the minimum distance only (of also the absolute value is 0)!

The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_pba.h

10.4 argus_meas_frame_t Struct Reference

The device measurement configuration structure.
`#include <argus_meas.h>`

Data Fields

- uint32_t PxEnMask
- uint32_t ChEnMask
- uq10_6_t AnalogIntegrationDepth
- uint16_t DigitalIntegrationDepth
- uq12_4_t OutputPower
- uint8_t PixelGain
- int8_t PllOffset
- argus_state_t State

10.4.1 Detailed Description

The device measurement configuration structure.
The portion of the configuration data that belongs to the measurement cycle. I.e. the data that defines a measurement frame.

10.4.2 Field Documentation

10.4.2.1 AnalogIntegrationDepth

`uq10_6_t argus_meas_frame_t::AnalogIntegrationDepth`

Pattern count per sample in uq10.6 format. Determines the analog integration depth.

10.4.2.2 ChEnMask

`uint32_t argus_meas_frame_t::ChEnMask`

ADS channel enabled mask for the remaining channels 31 .. 63 (miscellaneous values). See pixel mapping for more details on the channel mask.

10.4.2.3 DigitalIntegrationDepth

`uint16_t argus_meas_frame_t::DigitalIntegrationDepth`

Sample count per phase/frame. Determines the digital integration depth.

10.4.2.4 OutputPower

`uq12_4_t argus_meas_frame_t::OutputPower`
Laser current per sample in mA. Determines the optical output power.

10.4.2.5 PixelGain

`uint8_t argus_meas_frame_t::PixelGain`
Charge pump voltage per sample in LSB. Determines the pixel gain.

10.4.2.6 PllOffset

`int8_t argus_meas_frame_t::PllOffset`
PLL Frequency Offset, caused by temperature compensation, in PLL_INT_PRD LSBs.

10.4.2.7 PxEnMask

`uint32_t argus_meas_frame_t::PxEnMask`
ADC channel enabled mask for the first channels 0 .. 31 (active pixels channels). See pixel mapping for more details on the pixel mask.

10.4.2.8 State

`argus_state_t argus_meas_frame_t::State`
The current state of the measurement frame:

- Measurement Mode,
- A/B Frame,
- PLL_Locked Bit,
- BGL Warning/Error,
- DCA State,
- ...

The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_meas.h

10.5 argus_pixel_t Struct Reference

The evaluated measurement results per pixel.
`#include <argus_px.h>`

Data Fields

- q9_22_t Range
- uq1_15_t Phase
- uq12_4_t Amplitude
- argus_px_status_t Status
- int8_t RangeWindow
- uq12_4_t AmplitudeRaw

10.5.1 Detailed Description

The evaluated measurement results per pixel.
This structure contains the evaluated data for a single pixel.
If the amplitude is 0, the pixel is turned off or has invalid data.

10.5.2 Field Documentation

10.5.2.1 Amplitude

`uq12_4_t argus_pixel_t::Amplitude`
Amplitudes of measured signals in LSB. Special values: 0 == Pixel Off, 0xFFFF == Overflow/Error

10.5.2.2 AmplitudeRaw

`uq12_4_t argus_pixel_t::AmplitudeRaw`
The raw amplitudes of measured signals in LSB.

10.5.2.3 Phase

`uq1_15_t argus_pixel_t::Phase`
Phase Values from the device in units of PI, i.e. 0 ... 2.

10.5.2.4 Range

`q9_22_t argus_pixel_t::Range`
Range Values from the device in meter. It is the actual distance before software adjustments/calibrations.

10.5.2.5 RangeWindow

`int8_t argus_pixel_t::RangeWindow`
The unambiguous window determined by the dual frequency feature.

10.5.2.6 Status

`argus_px_status_t argus_pixel_t::Status`
Pixel status; determines if the pixel is disabled, saturated, .. See the pixel status flags for more information.
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_px.h

10.6 argus_results_aux_t Struct Reference

The auxiliary measurement results data structure.
`#include <argus_res.h>`

Data Fields

- uq12_4_t VDD
- q11_4_t TEMP
- uq12_4_t VSUB
- uq12_4_t VDDL
- uq12_4_t IAPD
- uq12_4_t BGL
- uq12_4_t SNA

10.6.1 Detailed Description

The auxiliary measurement results data structure.
The auxiliary measurement results obtained by the auxiliary task. Special values, i.e. 0xFFFFU, indicate no readout value available.

10.6.2 Field Documentation

10.6.2.1 BGL

uq12_4_t argus_results_aux_t::BGL
Background Light Value in arbitrary. units, estimated by the substrate voltage control task. Special Value if no value is available: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)

10.6.2.2 IAPD

uq12_4_t argus_results_aux_t::IAPD
APD current ADC Channel readout value. Special Value if no value has been measured: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)

10.6.2.3 SNA

uq12_4_t argus_results_aux_t::SNA
Shot Noise Amplitude in LSB units, estimated by the shot noise monitor task from the average amplitude of the passive pixels. Special Value if no value is available: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)

10.6.2.4 TEMP

q11_4_t argus_results_aux_t::TEMP
Temperature sensor ADC channel readout value. Special Value if no value has been measured: Invalid/NotAvailable = 0x7FFFU (Q11_4_MAX)

10.6.2.5 VDD

uq12_4_t argus_results_aux_t::VDD
VDD ADC channel readout value. Special Value if no value has been measured: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)

10.6.2.6 VDDL

uq12_4_t argus_results_aux_t::VDDL
VDD VCSEL ADC channel readout value. Special Value if no value has been measured: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)

10.6.2.7 VSUB

uq12_4_t argus_results_aux_t::VSUB
Substrate Voltage ADC Channel readout value. Special Value if no value has been measured: Invalid/NotAvailable = 0xFFFFU (UQ12_4_MAX)
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_res.h

10.7 argus_results_bin_t Struct Reference

The 1d measurement results data structure.
`#include <argus_res.h>`

Data Fields

- q9_22_t Range
- uq12_4_t Amplitude

10.7.1 Detailed Description

The 1d measurement results data structure.
The 1d measurement results obtained by the pixel binning algorithm.

10.7.2 Field Documentation

10.7.2.1 Amplitude

`uq12_4_t argus_results_bin_t::Amplitude`
The 1D amplitude in LSB (Q12.4 format). The (maximum) amplitude obtained by the pixel binning algorithm from the current measurement frame. Special value: 0 == No/Invalid Result.

10.7.2.2 Range

`q9_22_t argus_results_bin_t::Range`
Raw 1D range value in meter (Q9.22 format). The distance obtained by the pixel binning algorithm from the current measurement frame.

Examples

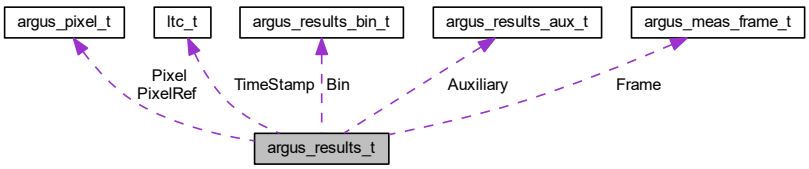
01_simple_example.c, and 02_advanced_example.c.

The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_res.h

10.8 argus_results_t Struct Reference

The measurement results data structure.
`#include <argus_res.h>`
Collaboration diagram for argus_results_t:



Data Fields

- status_t Status
- ltc_t TimeStamp
- argus_meas_frame_t Frame
- uint32_t Data [ARGUS_RAW_DATA_VALUES]
- argus_pixel_t PixelRef
- argus_pixel_t Pixel [ARGUS_PIXELS_X][ARGUS_PIXELS_Y]
- argus_results_bin_t Bin
- argus_results_aux_t Auxiliary

10.8.1 Detailed Description

The measurement results data structure.
Measurement data from the device.

```
// Pixel Field: Pixel[x][y]
//
// 0  ----------> x
// |  O O O O O O O O
// |   O O O O O O O O
// |  O O O O O O O O          O (ref. Px)
// y   O O O O O O O O
```

Examples

01_simple_example.c, and 02_advanced_example.c.

10.8.2 Field Documentation

10.8.2.1 Auxiliary

`argus_results_aux_t argus_results_t::Auxiliary`
The auxiliary ADC channel data.

10.8.2.2 Bin

`argus_results_bin_t argus_results_t::Bin`
Pixel binned results.

Examples

01_simple_example.c, and 02_advanced_example.c.

10.8.2.3 Data

`uint32_t argus_results_t::Data[ARGUS_RAW_DATA_VALUES]`
Raw unmapped ADC results from the device.

10.8.2.4 Frame

`argus_meas_frame_t argus_results_t::Frame`
The configuration for the current measurement frame.

10.8.2.5 Pixel

`argus_pixel_t argus_results_t::Pixel[ARGUS_PIXELS_X][ARGUS_PIXELS_Y]`
Raw Range Values from the device in meter. It is the actual distance before software adjustments/calibrations.

10.8.2.6 PixelRef

`argus_pixel_t argus_results_t::PixelRef`
Raw Range Values from the device in meter. It is the actual distance before software adjustments/calibrations.

10.8.2.7 Status

`status_t argus_results_t::Status`
The status of the current measurement frame.

- 0 (i.e. STATUS_OK) for a good measurement signal.
- $>$ 0 for warnings and weak measurement signal.
- $<$ 0 for errors and invalid measurement signal.

10.8.2.8 TimeStamp

`ltc_t argus_results_t::TimeStamp`
Time in milliseconds (measured since the last MCU startup/reset) when the measurement was triggered.
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_res.h

10.9 ltc_t Struct Reference

A data structure to represent current time.
`#include <time.h>`

Data Fields

- uint32_t sec
- uint32_t usec

10.9.1 Detailed Description

A data structure to represent current time.
Value is obtained from the PIT time which must be configured as lifetime counter.

10.9.2 Field Documentation

10.9.2.1 sec

`uint32_t ltc_t::sec`
Seconds.

10.9.2.2 usec

`uint32_t ltc_t::usec`
Microseconds.
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/utility/time.h

10.10 xtalk_t Struct Reference

Pixel Crosstalk Compensation Vector.
`#include <argus_xtalk.h>`

Data Fields

- q11_4_t dS
- q11_4_t dC

10.10.1 Detailed Description

Pixel Crosstalk Compensation Vector.
Contains calibration data (per pixel) that belongs to the RX-TX-Crosstalk compensation feature.
Pixel Crosstalk Vector

10.10.2 Field Documentation

10.10.2.1 dC

q11_4_t xtalk_t::dC
Crosstalk Vector - Cosine component. Special Value: Q11_4_MIN == not available

10.10.2.2 dS

q11_4_t xtalk_t::dS
Crosstalk Vector - Sine component. Special Value: Q11_4_MIN == not available
The documentation for this struct was generated from the following file:

- Sources/argus_api/include/api/argus_xtalk.h

Chapter 11

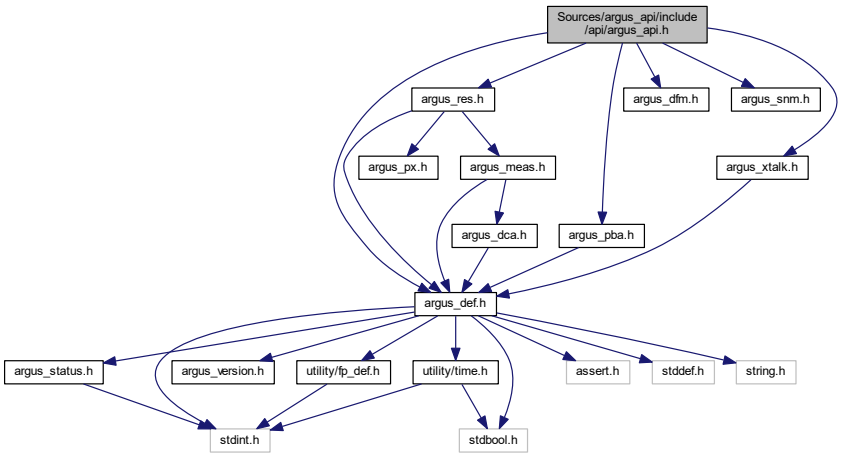
File Documentation

11.1 Sources/argus_api/include/api/argus_api.h File Reference

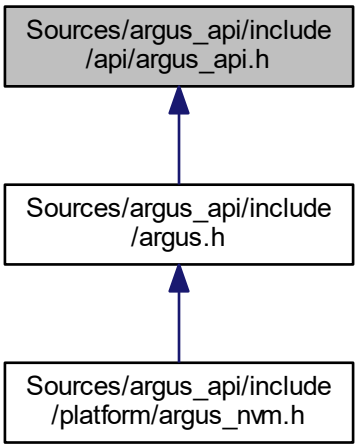
This file is part of the AFBR-S50 API.

```
#include "argus_def.h"
#include "argus_res.h"
#include "argus_pba.h"
#include "argus_dfm.h"
#include "argus_snm.h"
#include "argus_xtalk.h"
```

Include dependency graph for argus_api.h:



This graph shows which files directly or indirectly include this file:



Typedefs

- typedef struct Argus_Handle argus_hnd_t
- typedef int32_t s2pi_slave_t

Functions

- status_t Argus_Init (argus_hnd_t *hnd, s2pi_slave_t spi_slave)

  Initializes the API modules and the device with default parameters.
- status_t Argus_Reinit (argus_hnd_t *hnd)

  Reinitializes the API modules and the device with default parameters.
- status_t Argus_Deinit (argus_hnd_t *hnd)

  Deinitializes the API modules and the device.
- argus_hnd_t * Argus_CreateHandle (void)

  Creates a new device data handle object to store all internal states.
- void Argus_DestroyHandle (argus_hnd_t *hnd)

  Destroys a given device data handle object.
- uint32_t Argus_GetAPIVersion (void)

  Gets the version number of the current API library.
- char const * Argus_GetBuildNumber (void)

  Gets the build number of the current API library.
- argus_module_version_t Argus_GetModuleVersion (argus_hnd_t *hnd)

  Gets the version/variant of the module.
- argus_chip_version_t Argus_GetChipVersion (argus_hnd_t *hnd)

  Gets the version number of the chip.
- argus_laser_type_t Argus_GetLaserType (argus_hnd_t *hnd)

  Gets the type number of the device laser.
- uint32_t Argus_GetChipID (argus_hnd_t *hnd)

  Gets the unique identification number of the chip.
- s2pi_slave_t Argus_GetSPISlave (argus_hnd_t *hnd)

Gets the SPI hardware slave identifier.

- status_t Argus_StartMeasurementTimer (argus_hnd_t *hnd, argus_callback_t cb)

  Starts the timer based measurement cycle asynchronously.
- status_t Argus_StopMeasurementTimer (argus_hnd_t *hnd)

  Stops the timer based measurement cycle.
- status_t Argus_TriggerMeasurement (argus_hnd_t *hnd, argus_callback_t cb)

  Triggers a single measurement frame asynchronously.
- status_t Argus_Abort (argus_hnd_t *hnd)

  Stops the currently ongoing measurements and SPI activity immediately.
- status_t Argus_GetStatus (argus_hnd_t *hnd)

  Checks the state of the device/driver.
- status_t Argus_Ping (argus_hnd_t *hnd)

  Tests the connection to the device by sending a ping message.
- status_t Argus_EvaluateData (argus_hnd_t *hnd, argus_results_t *res, void *raw)

  Evaluate useful information from the raw measurement data.
- status_t Argus_ExecuteXtalkCalibrationSequence (argus_hnd_t *hnd, argus_mode_t mode)

  Executes a crosstalk calibration measurement.
- status_t Argus_ExecuteRelativeRangeOffsetCalibrationSequence (argus_hnd_t *hnd, argus_mode_t mode)

  Executes a relative range offset calibration measurement.
- status_t Argus_ExecuteAbsoluteRangeOffsetCalibrationSequence (argus_hnd_t *hnd, argus_mode_t mode, q9_22_t targetRange)

  Executes an absolute range offset calibration measurement.
- status_t Argus_SetConfigurationMeasurementMode (argus_hnd_t *hnd, argus_mode_t value)

  Sets the measurement mode to a specified device.
- status_t Argus_GetConfigurationMeasurementMode (argus_hnd_t *hnd, argus_mode_t *value)

  Gets the measurement mode from a specified device.
- status_t Argus_SetConfigurationFrameTime (argus_hnd_t *hnd, uint32_t value)

  Sets the frame time to a specified device.
- status_t Argus_GetConfigurationFrameTime (argus_hnd_t *hnd, uint32_t *value)

  Gets the frame time from a specified device.
- status_t Argus_SetConfigurationSmartPowerSaveEnabled (argus_hnd_t *hnd, argus_mode_t mode, bool value)

  Sets the smart power save enabled flag to a specified device.
- status_t Argus_GetConfigurationSmartPowerSaveEnabled (argus_hnd_t *hnd, argus_mode_t mode, bool *value)

  Gets the smart power save enabled flag from a specified device.
- status_t Argus_SetConfigurationDFMMode (argus_hnd_t *hnd, argus_mode_t mode, argus_dfm_mode_t value)

  Sets the Dual Frequency Mode (DFM) to a specified device.
- status_t Argus_GetConfigurationDFMMode (argus_hnd_t *hnd, argus_mode_t mode, argus_dfm_mode_t *value)

  Gets the Dual Frequency Mode (DFM) from a specified device.
- status_t Argus_SetConfigurationShotNoiseMonitorMode (argus_hnd_t *hnd, argus_mode_t mode, argus_snm_mode_t value)

  Sets the Shot Noise Monitor (SNM) mode to a specified device.
- status_t Argus_GetConfigurationShotNoiseMonitorMode (argus_hnd_t *hnd, argus_mode_t mode, argus_snm_mode_t *value)

  Gets the Shot Noise Montor (SNM) mode from a specified device.
- status_t Argus_SetConfigurationDynamicAdaption (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_dca_t const *value)

  Sets the full DCA module configuration to a specified device.

- status_t Argus_GetConfigurationDynamicAdaption (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_dca_t *value)

  Gets the # from a specified device.
- status_t Argus_SetConfigurationPixelBinning (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_pba_t const *value)

  Sets the pixel binning configuration parameters to a specified device.
- status_t Argus_GetConfigurationPixelBinning (argus_hnd_t *hnd, argus_mode_t mode, argus_cfg_pba_t *value)

  Gets the pixel binning configuration parameters from a specified device.
- status_t Argus_GetConfigurationUnambiguousRange (argus_hnd_t *hnd, uint32_t *range_mm)

  Gets the current unambiguous range in mm.
- status_t Argus_SetCalibrationGlobalRangeOffset (argus_hnd_t *hnd, argus_mode_t mode, q9_22_t value)

  Sets the global range offset value to a specified device.
- status_t Argus_GetCalibrationGlobalRangeOffset (argus_hnd_t *hnd, argus_mode_t mode, q9_22_t *value)

  Gets the global range offset value from a specified device.
- status_t Argus_SetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Sets the relative pixel offset table to a specified device.
- status_t Argus_GetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the relative pixel offset table from a specified device.
- status_t Argus_GetCalibrationTotalPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode, q0_15_t value[ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the relative pixel offset table from a specified device.
- status_t Argus_ResetCalibrationPixelRangeOffsets (argus_hnd_t *hnd, argus_mode_t mode)

  Resets the relative pixel offset values for the specified device to the factory calibrated default values.
- void Argus_GetExternalPixelRangeOffsets_Callback (q0_15_t offsets[ARGUS_PIXELS_X][ARGUS_PIXELS_Y], argus_mode_t mode)

  A callback that returns the external pixel range offsets.
- status_t Argus_SetCalibrationRangeOffsetSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the range offset calibration sequence.
- status_t Argus_GetCalibrationRangeOffsetSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the range offset calibration sequence.
- status_t Argus_SetCalibrationCrosstalkPixel2Pixel (argus_hnd_t *hnd, argus_mode_t mode, argus_cal_p2pxtalk_t const *value)

  Sets the pixel-to-pixel crosstalk compensation parameters to a specified device.
- status_t Argus_GetCalibrationCrosstalkPixel2Pixel (argus_hnd_t *hnd, argus_mode_t mode, argus_cal_p2pxtalk_t *value)

  Gets the pixel-to-pixel crosstalk compensation parameters from a specified device.
- status_t Argus_SetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Sets the custom crosstalk vector table to a specified device.
- status_t Argus_GetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the custom crosstalk vector table from a specified device.
- status_t Argus_GetCalibrationTotalCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode, xtalk_t value[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][ARGUS_PIXELS_Y])

  Gets the factory calibrated default crosstalk vector table for the specified device.
- status_t Argus_ResetCalibrationCrosstalkVectorTable (argus_hnd_t *hnd, argus_mode_t mode)

  Resets the crosstalk vector table for the specified device to the factory calibrated default values.
- status_t Argus_SetCalibrationCrosstalkSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the crosstalk calibration sequence.

- status_t Argus_GetCalibrationCrosstalkSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the crosstalk calibration sequence.

- status_t Argus_SetCalibrationCrosstalkSequenceAmplitudeThreshold (argus_hnd_t *hnd, uq12_4_t value)

  Sets the max. amplitude threshold for the crosstalk calibration sequence.

- status_t Argus_GetCalibrationCrosstalkSequenceAmplitudeThreshold (argus_hnd_t *hnd, uq12_4_t *value)

  Gets the max. amplitude threshold for the crosstalk calibration sequence.

- status_t Argus_SetCalibrationVsubSequenceSampleCount (argus_hnd_t *hnd, uint16_t value)

  Sets the sample count for the substrate voltage calibration sequence.

- status_t Argus_GetCalibrationVsubSequenceSampleCount (argus_hnd_t *hnd, uint16_t *value)

  Gets the sample count for the substrate voltage calibration sequence.

- void Argus_GetExternalCrosstalkVectorTable_Callback (xtalk_t xtalk[ARGUS_DFM_FRAME_COUNT][ARGUS_PIXELS_X][AF argus_mode_t mode)

  A callback that returns the external crosstalk vector table.

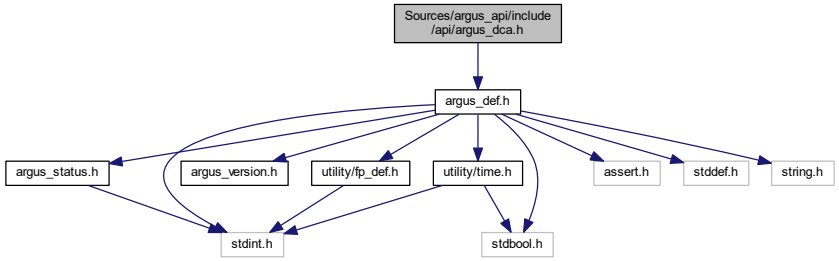
11.1.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides generic functionality belonging to all devices from the AFBR-S50 product family.

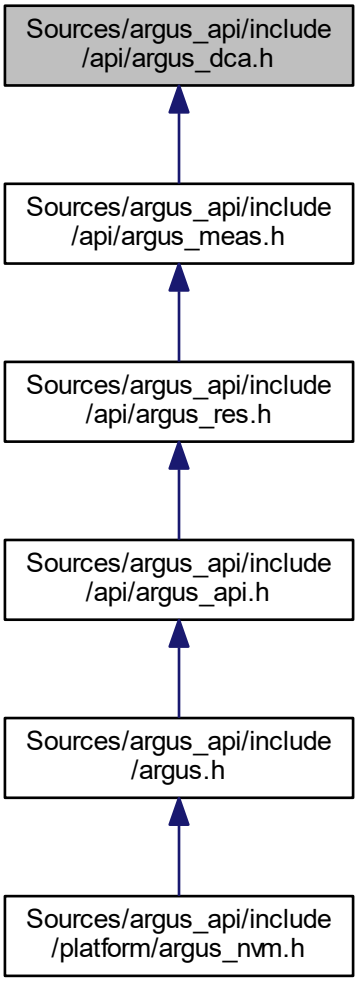
Copyright

11.2 Sources/argus_api/include/api/argus_dca.h File Reference

This file is part of the AFBR-S50 API.
`#include "argus_def.h"`
Include dependency graph for argus_dca.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct argus_cfg_dca_t

  Dynamic Configuration Adaption (DCA) Parameters.

Macros

- #define ARGUS_CFG_DCA_ATH_MIN (1U << 6U)
- #define ARGUS_CFG_DCA_ATH_MAX (0xFFFFU)
- #define ARGUS_CFG_DCA_PXTH_MIN (1U)
- #define ARGUS_CFG_DCA_PXTH_MAX (33U)
- #define ARGUS_CFG_DCA_DEPTH_MAX ((uq10_6_t)(ADS_SEQCT_N_MASK << (6U - ADS_SEQCT↩ _N_SHIFT)))
- #define ARGUS_CFG_DCA_DEPTH_MIN ((uq10_6_t)(1U))
- #define ARGUS_CFG_DCA_POWER_MAX_LSB (ADS_LASET_VCSEL_HC1_MASK >> ADS_LASET_↩ VCSEL_HC1_SHIFT)

- #define ARGUS_CFG_DCA_POWER_MIN_LSB (1)
- #define ARGUS_CFG_DCA_POWER_MAX (ADS0032_HIGH_CURRENT_LSB2MA(ARGUS_CFG_DCA_POWER_MAX_LSB + 1))
- #define ARGUS_CFG_DCA_POWER_MIN (1)
- #define ARGUS_DCA_GAIN_STAGE_COUNT (4U)
- #define ARGUS_STATE_DCA_GAIN_MASK (0x03U)
- #define ARGUS_STATE_DCA_GAIN_SHIFT (14U)
- #define ARGUS_STATE_DCA_GAIN_GET(state) (((state) >> ARGUS_STATE_DCA_GAIN_SHIFT) & ARGUS_STATE_DCA_GAIN_MASK)
- #define ARGUS_DCA_POWER_STAGE_COUNT (4U)
- #define ARGUS_STATE_DCA_POWER_MASK (0x03U)
- #define ARGUS_STATE_DCA_POWER_SHIFT (12U)
- #define ARGUS_STATE_DCA_POWER_GET(state) (((state) >> ARGUS_STATE_DCA_POWER_SHIFT) & ARGUS_STATE_DCA_POWER_MASK)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_MASK (0x0FU)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT (8U)
- #define ARGUS_STATE_DCA_DEPTH_SHFT_GET(state) (((state) >> ARGUS_STATE_DCA_DEPTH_SHFT_SHIFT) & ARGUS_STATE_DCA_DEPTH_SHFT_MASK)

Enumerations

- enum argus_dca_enable_t {
  DCA_ENABLE_OFF = 0,
  DCA_ENABLE_DYNAMIC = 1,
  DCA_ENABLE_STATIC = -1 }

  The dynamic configuration algorithm enable flags.
- enum argus_dca_power_t {
  DCA_POWER_LOW = 0,
  DCA_POWER_MEDIUM_LOW = 1,
  DCA_POWER_MEDIUM_HIGH = 2,
  DCA_POWER_HIGH = 3 }

  The dynamic configuration algorithm Optical Output Power stages enumerator.
- enum argus_dca_gain_t {
  DCA_GAIN_LOW = 0,
  DCA_GAIN_MEDIUM_LOW = 1,
  DCA_GAIN_MEDIUM_HIGH = 2,
  DCA_GAIN_HIGH = 3 }

  The dynamic configuration algorithm Pixel Input Gain stages enumerator.
- enum argus_state_t {
  ARGUS_STATE_NONE = 0,
  ARGUS_STATE_MEASUREMENT_MODE = 1U << 0U,
  ARGUS_STATE_DUAL_FREQ_MODE = 1U << 1U,
  ARGUS_STATE_MEASUREMENT_FREQ = 1U << 2U,
  ARGUS_STATE_DEBUG_MODE = 1U << 3U,
  ARGUS_STATE_GOLDEN_PIXEL_MODE = 1U << 4U,
  ARGUS_STATE_BGL_WARNING = 1U << 5U,
  ARGUS_STATE_BGL_ERROR = 1U << 6U,
  ARGUS_STATE_PLL_LOCKED = 1U << 7U,
  ARGUS_STATE_DCA_POWER_LOW = DCA_GAIN_LOW << ARGUS_STATE_DCA_POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_MED_LOW = DCA_GAIN_MEDIUM_LOW << ARGUS_STATE_DCA_↩ POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_MED_HIGH = DCA_GAIN_MEDIUM_HIGH << ARGUS_STATE_DCA_↩ POWER_SHIFT,
  ARGUS_STATE_DCA_POWER_HIGH = DCA_GAIN_HIGH << ARGUS_STATE_DCA_POWER_SHIFT,
  ARGUS_STATE_DCA_GAIN_LOW = DCA_GAIN_LOW << ARGUS_STATE_DCA_GAIN_SHIFT,
  ARGUS_STATE_DCA_GAIN_MED_LOW = DCA_GAIN_MEDIUM_LOW << ARGUS_STATE_DCA_GAI↩ N_SHIFT,

ARGUS_STATE_DCA_GAIN_MED_HIGH = DCA_GAIN_MEDIUM_HIGH << ARGUS_STATE_DCA_G↩
AIN_SHIFT,
ARGUS_STATE_DCA_GAIN_HIGH = DCA_GAIN_HIGH << ARGUS_STATE_DCA_GAIN_SHIFT }

State flags for the current frame.

11.2.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the dynamic configuration adaption (DCA) setup parameters and data structure.

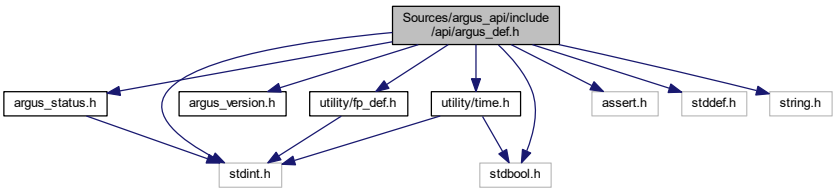
Copyright

11.3 Sources/argus_api/include/api/argus_def.h File Reference

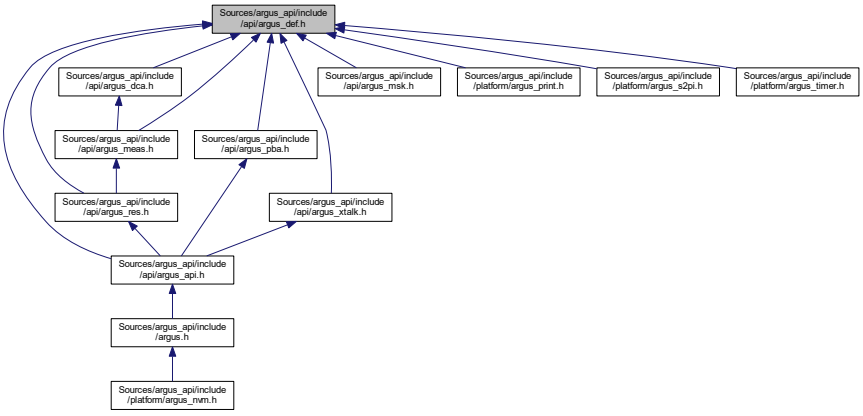
This file is part of the AFBR-S50 hardware API.

```
#include "argus_status.h"
#include "argus_version.h"
#include "utility/fp_def.h"
#include "utility/time.h"
#include <assert.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <string.h>
```

Include dependency graph for argus_def.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define ARGUS_PHASECOUNT 4U

  Maximum number of phases per measurement cycle.
- #define ARGUS_PIXELS_X 8U

  The device pixel field size in x direction (long edge).
- #define ARGUS_PIXELS_Y 4U

  The device pixel field size in y direction (short edge).
- #define ARGUS_PIXELS ((ARGUS_PIXELS_X)∗(ARGUS_PIXELS_Y))

  The total device pixel count.
- #define ARGUS_MODE_COUNT (2)

  The number of measurement modes with distinct configuration and calibration records.

Typedefs

- typedef status_t(∗ argus_callback_t) (status_t status, void ∗data)

  Generic API callback function.

Enumerations

- enum argus_module_version_t {
  MODULE_NONE = 0,
  AFBR_S50MV85G_V1 = 1,
  AFBR_S50MV85G_V2 = 2,
  AFBR_S50MV85G_V3 = 7,
  AFBR_S50LV85D_V1 = 3,
  AFBR_S50MV68B_V1 = 4,
  AFBR_S50MV85I_V1 = 5,
  AFBR_S50SV85K_V1 = 6,
  Reserved = 0b111111 }

  The AFBR-S50 module types.
- enum argus_laser_type_t {
  LASER_NONE = 0,
  LASER_H_V1 = 1,
  LASER_H_V2 = 2,
  LASER_R_V1 = 3 }

  The AFBR-S50 laser configurations.
- enum argus_chip_version_t {
  ADS0032_NONE = 0,
  ADS0032_V1_0 = 1,
  ADS0032_V1_1 = 2,
  ADS0032_V1_2 = 3 }

  The AFBR-S50 chip versions.
- enum argus_mode_t {
  ARGUS_MODE_A = 1,
  ARGUS_MODE_B = 2 }

  The measurement modes.

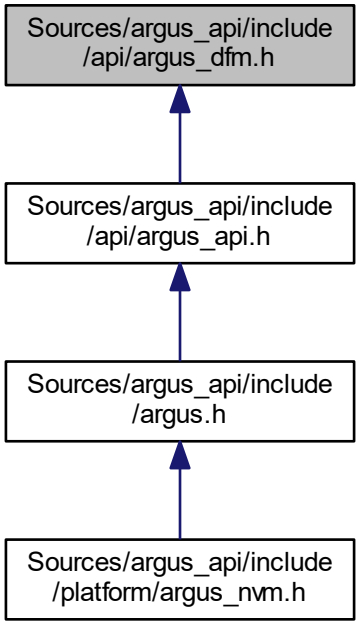
11.3.1 Detailed Description

This file is part of the AFBR-S50 hardware API.
This file provides generic definitions belonging to all devices from the AFBR-S50 product family.

Copyright

11.4 Sources/argus_api/include/api/argus_dfm.h File Reference

This file is part of the AFBR-S50 API.
This graph shows which files directly or indirectly include this file:



Macros

- #define ARGUS_DFM_FRAME_COUNT (2U)
- #define ARGUS_DFM_MODE_COUNT (2U)

Enumerations

- enum argus_dfm_mode_t {
  DFM_MODE_OFF = 0U,
  DFM_MODE_4X = 1U,
  DFM_MODE_8X = 2U }

11.4.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the dual frequency mode (DFM) setup parameters.

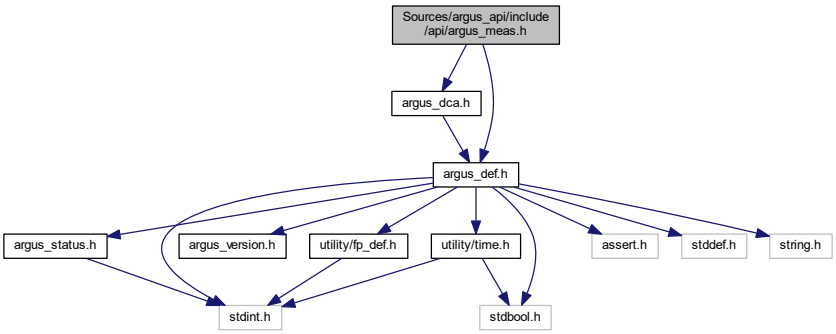
Copyright

11.5 Sources/argus_api/include/api/argus_meas.h File Reference

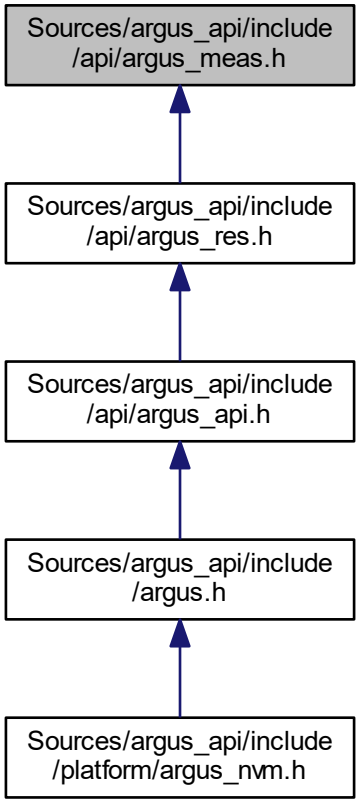
This file is part of the AFBR-S50 hardware API.

```
#include "argus_dca.h"
#include "argus_def.h"
```

Include dependency graph for argus_meas.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct argus_meas_frame_t

The device measurement configuration structure.

Macros

- #define ARGUS_RAW_DATA_VALUES 132U
- #define ARGUS_RAW_DATA_SIZE (3U ∗ ARGUS_RAW_DATA_VALUES)
- #define ARGUS_AUX_CHANNEL_COUNT (5U)
- #define ARGUS_AUX_DATA_SIZE (3U ∗ ARGUS_AUX_CHANNEL_COUNT)

11.5.1 Detailed Description

This file is part of the AFBR-S50 hardware API.
Defines the generic measurement parameters and data structures.

Copyright

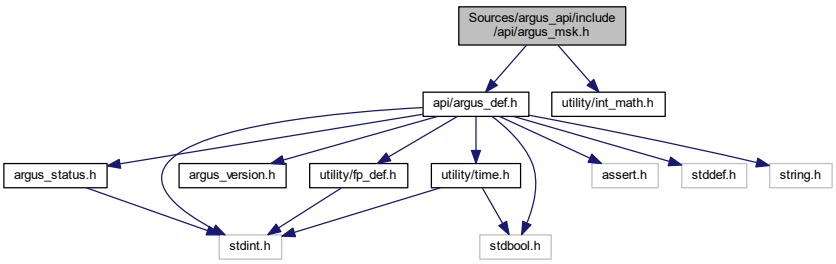
Copyright (c) 2016-2019, Avago Technologies GmbH. All rights reserved.

11.6 Sources/argus_api/include/api/argus_msk.h File Reference

This file is part of the AFBR-S50 API.

```
#include "api/argus_def.h"
#include "utility/int_math.h"
```

Include dependency graph for argus_msk.h:



Macros

- #define PIXEL_XY2N(x, y) ((((x) ^ 7) << 1) | ((y) & 2) << 3 | ((y) & 1))

  Macro to determine the channel number of an specified Pixel.
- #define PIXEL_N2X(n) ((((n) >> 1U) & 7) ^ 7)

  Macro to determine the x index of an specified Pixel channel.
- #define PIXEL_N2Y(n) (((n) & 1U) | (((n) >> 3) & 2U))

  Macro to determine the y index of an specified Pixel channel.
- #define PIXELN_ISENABLED(msk, ch) (((msk) >> (ch)) & 0x01U)

  Macro to determine if a ADC Pixel channel was enabled from a pixel mask.
- #define PIXELN_ENABLE(msk, ch) ((msk) |= (0x01U << (ch)))

  Macro enables an ADC Pixel channel in a pixel mask.
- #define PIXELN_DISABLE(msk, ch) ((msk) &= (∼(0x01U << (ch))))

  Macro disables an ADC Pixel channel in a pixel mask.
- #define PIXELXY_ISENABLED(msk, x, y) (PIXELN_ISENABLED(msk, PIXEL_XY2N(x, y)))

  Macro to determine if an ADC Pixel channel was enabled from a pixel mask.
- #define PIXELXY_ENABLE(msk, x, y) (PIXELN_ENABLE(msk, PIXEL_XY2N(x, y)))

Macro enables an ADC Pixel channel in a pixel mask.

- #define PIXELXY_DISABLE(msk, x, y) (PIXELN_DISABLE(msk, PIXEL_XY2N(x, y)))

  Macro disables an ADC Pixel channel in a pixel mask.

- #define CHANNELN_ISENABLED(msk, ch) (((msk) >> ((ch) - 32U)) & 0x01U)

  Macro to determine if a ADC channel was enabled from a channel mask.

- #define CHANNELN_ENABLE(msk, ch) ((msk) |= (0x01U << ((ch) - 32U)))

  Macro to determine if a ADC channel was enabled from a channel mask.

- #define CHANNELN_DISABLE(msk, ch) ((msk) &= (~(0x01U << ((ch) - 32U))))

  Macro to determine if a ADC channel was enabled from a channel mask.

- #define PIXEL_COUNT(pxmsk) popcount(pxmsk)

  Macro to determine the number of enabled pixel channels via a popcount algorithm.

- #define CHANNEL_COUNT(pxmsk, chmsk) (popcount(pxmsk) + popcount(chmsk))

  Macro to determine the number of enabled channels via a popcount algorithm.

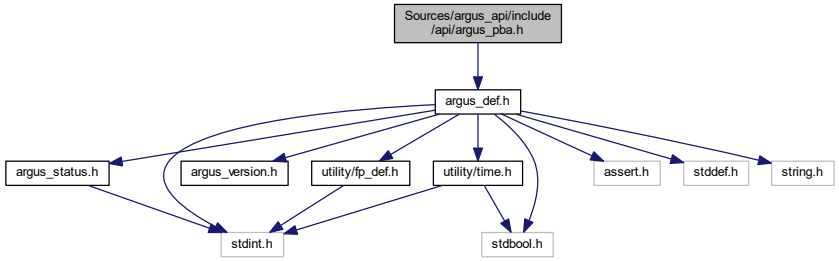
11.6.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines macros to work with pixel and ADC channel masks.

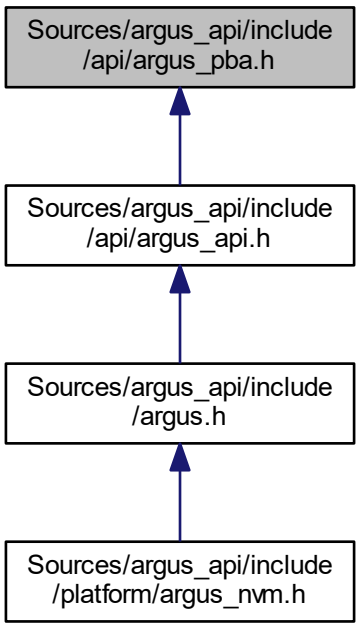
Copyright

11.7 Sources/argus_api/include/api/argus_pba.h File Reference

This file is part of the AFBR-S50 API.
`#include "argus_def.h"`
Include dependency graph for argus_pba.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct argus_cfg_pba_t

  The pixel binning algorithm settings data structure.

Enumerations

- enum argus_pba_flags_t {
  PBA_ENABLE = 1U << 0U,
  PBA_ENABLE_GOLDPX = 1U << 5U,
  PBA_ENABLE_MIN_DIST_SCOPE = 1U << 6U }

  Enable flags for the pixel binning algorithm.

- enum argus_pba_averaging_mode_t {
  PBA_SIMPLE_AVG = 1U,
  PBA_LINEAR_AMPLITUDE_WEIGHTED_AVG = 2U }

  The averaging modes for the pixel binning algorithm.

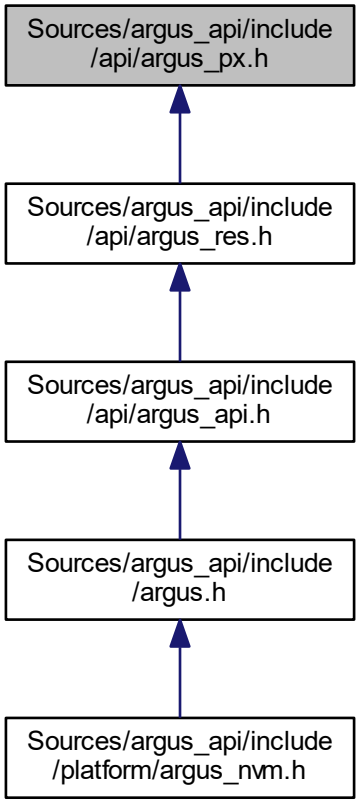
11.7.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the pixel binning algorithm (PBA) setup parameters and data structure.

Copyright

11.8 Sources/argus_api/include/api/argus_px.h File Reference

This file is part of the AFBR-S50 API.
This graph shows which files directly or indirectly include this file:



Data Structures

- struct argus_pixel_t

  The evaluated measurement results per pixel.

Macros

- #define ARGUS_AMPLITUDE_MAX (0xFFF0U)
- #define ARGUS_RANGE_MAX (Q9_22_MAX)

Enumerations

- enum argus_px_status_t {
  PIXEL_OK = 0,
  PIXEL_OFF = 1U << 0U,
  PIXEL_SAT = 1U << 1U,
  PIXEL_BIN_EXCL = 1U << 2U,
  PIXEL_AMPL_MIN = 1U << 3U,
  PIXEL_PREFILTERED = 1U << 4U,

PIXEL_NO_SIGNAL = 1U << 5U,
PIXEL_OUT_OF_SYNC = 1U << 6U,
PIXEL_STALLED = 1U << 7U }

Status flags for the evaluated pixel structure.

11.8.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the device pixel measurement results data structure.

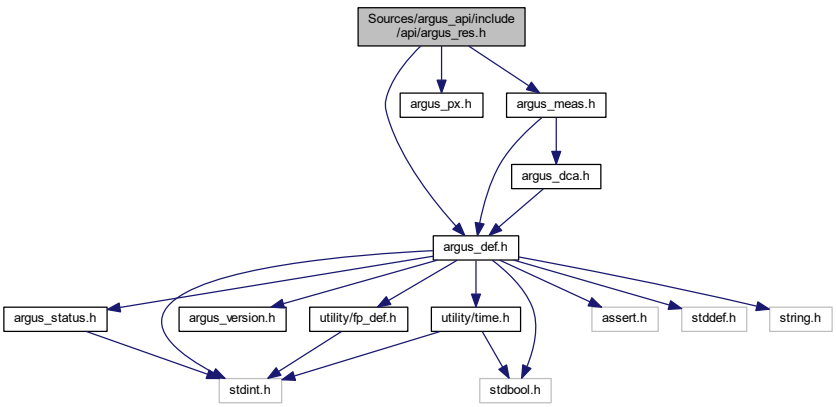
Copyright

11.9 Sources/argus_api/include/api/argus_res.h File Reference

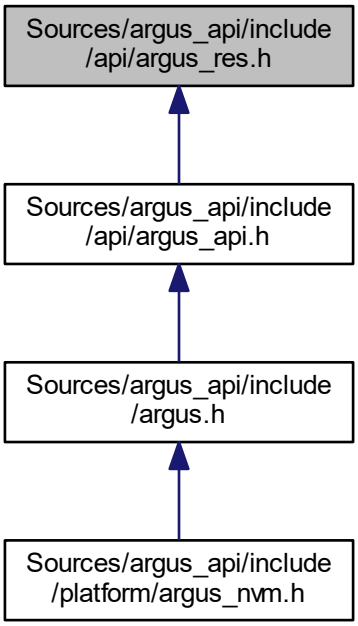
This file is part of the AFBR-S50 API.

```
#include "argus_def.h"
#include "argus_px.h"
#include "argus_meas.h"
```

Include dependency graph for argus_res.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct argus_results_bin_t

  The 1d measurement results data structure.

- struct argus_results_aux_t

  The auxiliary measurement results data structure.

- struct argus_results_t

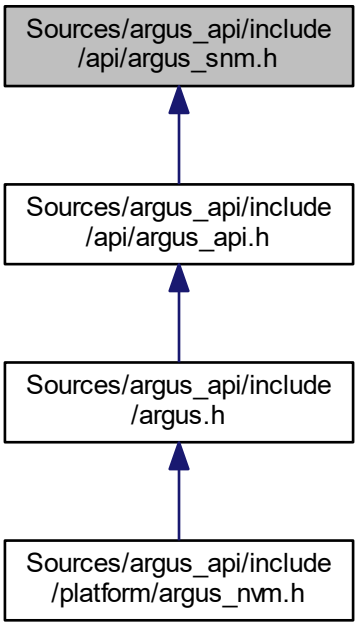
  The measurement results data structure.

11.9.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the generic measurement results data structure.

11.10 Sources/argus_api/include/api/argus_snm.h File Reference

This file is part of the AFBR-S50 API.
This graph shows which files directly or indirectly include this file:



Enumerations

- enum argus_snm_mode_t {
  SNM_MODE_STATIC_INDOOR = 0U,
  SNM_MODE_STATIC_OUTDOOR = 1U,
  SNM_MODE_DYNAMIC = 2U }

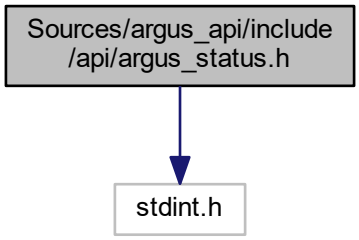
11.10.1 Detailed Description

This file is part of the AFBR-S50 API.
Defines the Shot Noise Monitor (SNM) setup parameters.

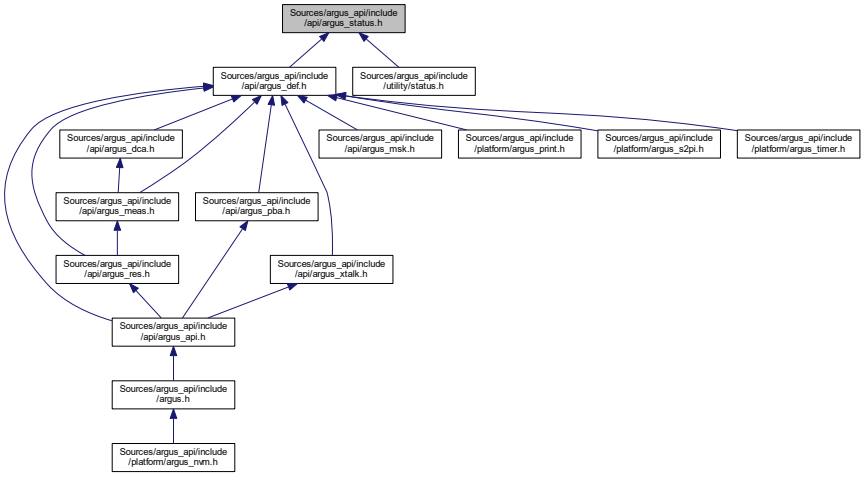
11.11 Sources/argus_api/include/api/argus_status.h File Reference

This file is part of the AFBR-S50 API.

`#include <stdint.h>`
Include dependency graph for argus_status.h:



This graph shows which files directly or indirectly include this file:



Typedefs

- typedef int32_t status_t

  Type used for all status and error return values.

Enumerations

- enum Status {
  STATUS_OK = 0,
  STATUS_IDLE = 0,
  STATUS_IGNORE = 1,
  STATUS_BUSY = 2,
  STATUS_INITIALIZING = 3,
  ERROR_FAIL = -1,
  ERROR_ABORTED = -2,
  ERROR_READ_ONLY = -3,
  ERROR_OUT_OF_RANGE = -4,
  ERROR_INVALID_ARGUMENT = -5,

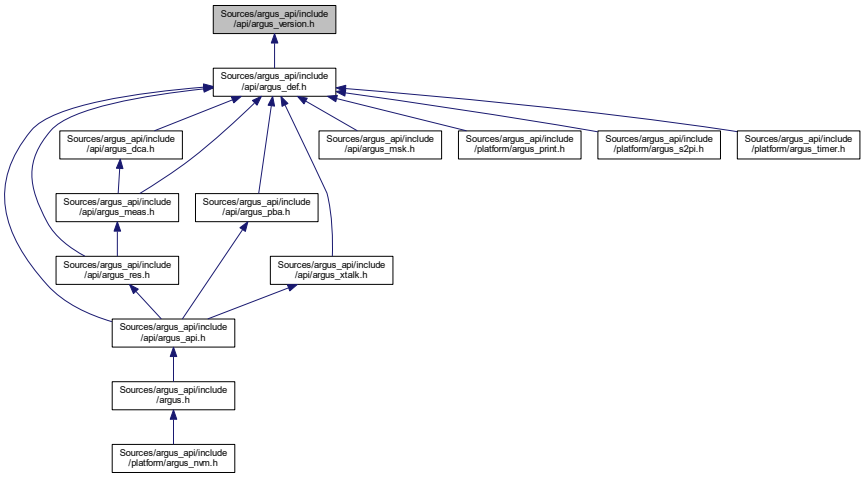
ERROR_TIMEOUT = -6,
ERROR_NOT_INITIALIZED = -7,
ERROR_NOT_SUPPORTED = -8,
ERROR_NOT_IMPLEMENTED = -9,
STATUS_S2PI_GPIO_MODE = 51,
ERROR_S2PI_RX_ERROR = -51,
ERROR_S2PI_TX_ERROR = -52,
ERROR_S2PI_INVALID_STATE = -53,
ERROR_S2PI_INVALID_BAUD_RATE = -54,
ERROR_S2PI_INVALID_SLAVE = -55,
ERROR_NVM_INVALID_FILE_VERSION = -98,
ERROR_NVM_OUT_OF_RANGE = -99,
STATUS_ARGUS_BUFFER_BUSY = 104,
STATUS_ARGUS_POWERLIMIT = 105,
STATUS_ARGUS_UNDERFLOW = 107,
STATUS_ARGUS_NO_OBJECT = 108,
STATUS_ARGUS_EEPROM_BIT_ERROR = 109,
STATUS_ARGUS_INVALID_EEPROM = 110,
ERROR_ARGUS_NOT_CONNECTED = -101,
ERROR_ARGUS_INVALID_CFG = -102,
ERROR_ARGUS_INVALID_MODE = -105,
ERROR_ARGUS_BIAS_VOLTAGE_REINIT = -107,
ERROR_ARGUS_EEPROM_FAILURE = -109,
ERROR_ARGUS_STALLED = -110,
ERROR_ARGUS_BGL_EXCEEDANCE = -111,
ERROR_ARGUS_XTALK_AMPLITUDE_EXCEEDANCE = -112,
ERROR_ARGUS_LASER_FAILURE = -113,
ERROR_ARGUS_DATA_INTEGRITY_LOST = -114,
ERROR_ARGUS_RANGE_OFFSET_CALIBRATION_FAILED = -115,
ERROR_ARGUS_BUSY = -191,
ERROR_ARGUS_UNKNOWN_MODULE = -199,
ERROR_ARGUS_UNKNOWN_CHIP = -198,
ERROR_ARGUS_UNKNOWN_LASER = -197,
STATUS_ARGUS_BUSY_CFG_UPDATE = 193,
STATUS_ARGUS_BUSY_CAL_UPDATE = 194,
STATUS_ARGUS_BUSY_CAL_SEQ = 195,
STATUS_ARGUS_BUSY_MEAS = 196,
STATUS_ARGUS_STARTING = 100,
STATUS_ARGUS_ACTIVE = 103 }

11.11.1 Detailed Description

This file is part of the AFBR-S50 API.
Provides status codes for the AFBR-S50 API.

11.12 Sources/argus_api/include/api/argus_version.h File Reference

This file is part of the AFBR-S50 API.
This graph shows which files directly or indirectly include this file:



Macros

- #define ARGUS_API_VERSION_MAJOR 1
- #define ARGUS_API_VERSION_MINOR 2
- #define ARGUS_API_VERSION_BUGFIX 3
- #define ARGUS_API_VERSION_BUILD "20201120091253"
- #define MAKE_VERSION(major, minor, bugfix) (((major) << 24) | ((minor) << 16) | (bugfix))
- #define ARGUS_API_VERSION

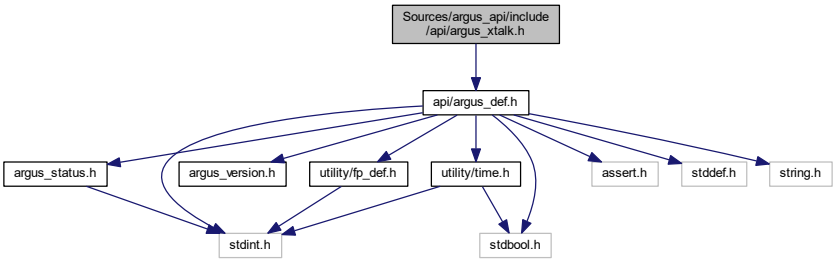
11.12.1 Detailed Description

This file is part of the AFBR-S50 API.
This file contains the current API version number.

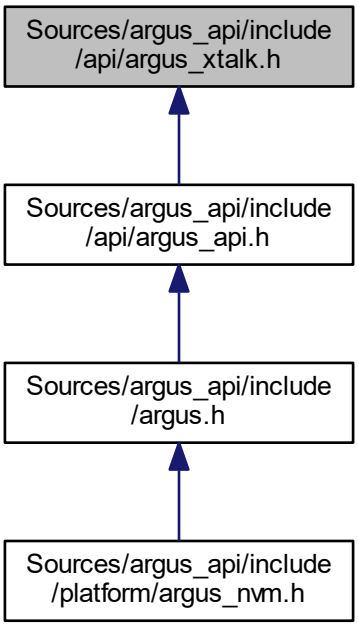
Copyright

11.13 Sources/argus_api/include/api/argus_xtalk.h File Reference

This file is part of the AFBR-S50 hardware API.
`#include "api/argus_def.h"`
Include dependency graph for argus_xtalk.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct xtalk_t

  Pixel Crosstalk Compensation Vector.
- struct argus_cal_p2pxtalk_t

Pixel-To-Pixel Crosstalk Compensation Parameters.

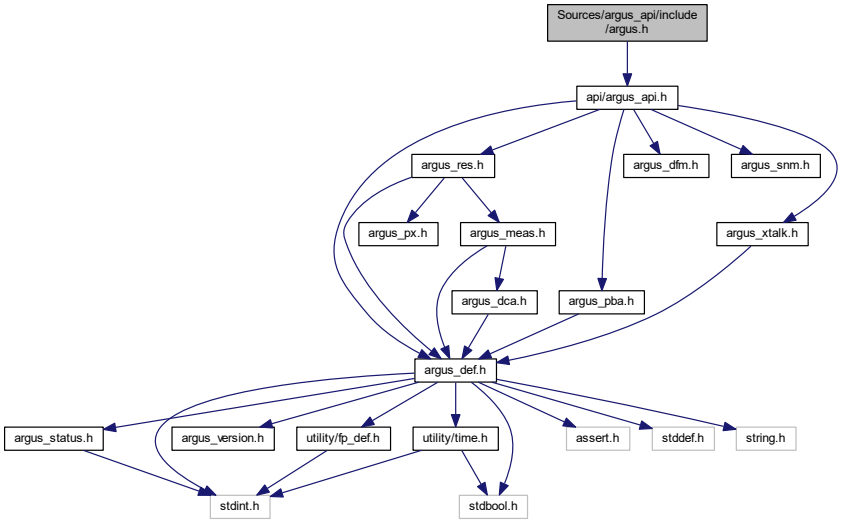
11.13.1 Detailed Description

This file is part of the AFBR-S50 hardware API.
Defines the generic device calibration API.

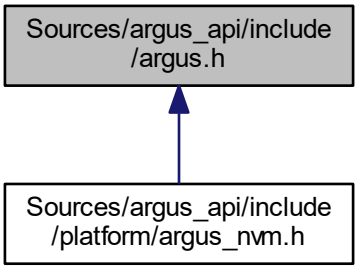
Copyright

11.14 Sources/argus_api/include/argus.h File Reference

This file is part of the AFBR-S50 API.
`#include "api/argus_api.h"`
Include dependency graph for argus.h:



This graph shows which files directly or indirectly include this file:



11.14.1 Detailed Description

This file is part of the AFBR-S50 API.
This file the main header of the AFBR-S50 API.

Copyright

11.15 Sources/argus_api/include/platform/argus_irq.h File Reference

This file is part of the AFBR-S50 API.

Functions

- void IRQ_UNLOCK (void)

  Enable IRQ Interrupts.
- void IRQ_LOCK (void)

  Disable IRQ Interrupts.

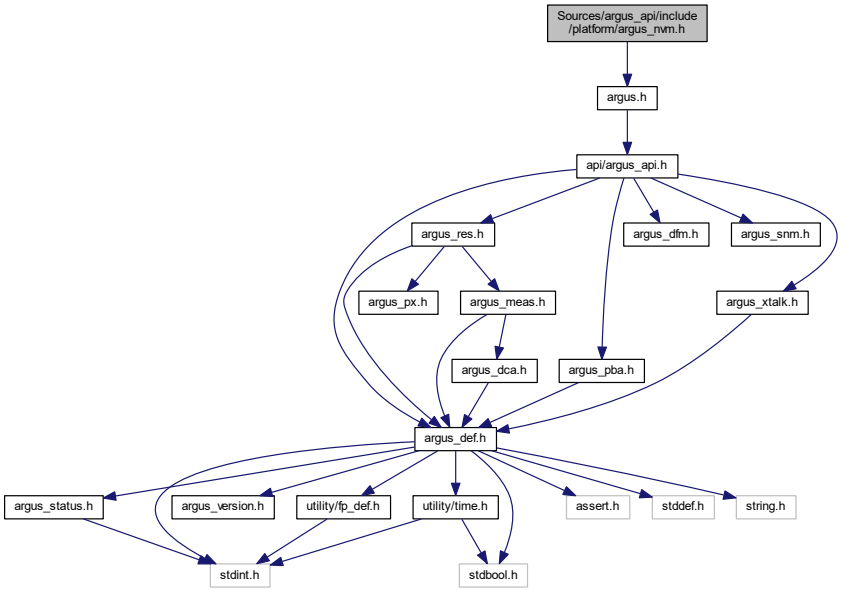
11.15.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides an interface for enabling/disabling interrupts.

Copyright

11.16 Sources/argus_api/include/platform/argus_nvm.h File Reference

This file is part of the AFBR-S50 API.
`#include "argus.h"`
Include dependency graph for argus_nvm.h:



Functions

- status_t NVM_Init (uint32_t size)

  Initializes the non-volatile memory unit and reserves a chunk of memory.
- status_t NVM_Write (uint32_t offset, uint32_t size, uint8_t const ∗buf)

  Write a block of data to the non-volatile memory.
- status_t NVM_Read (uint32_t offset, uint32_t size, uint8_t ∗buf)

  Reads a block of data from the non-volatile memory.

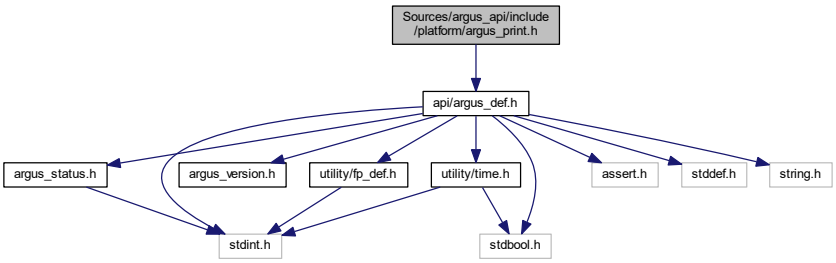
11.16.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides an interface for the optional non-volatile memory.

Copyright

11.17 Sources/argus_api/include/platform/argus_print.h File Reference

This file is part of the AFBR-S50 API.
`#include "api/argus_def.h"`
Include dependency graph for argus_print.h:



Functions

- status_t print (const char ∗fmt_s,...)

  A printf-like function to print formated data to an debugging interface.

11.17.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides an interface for the optional debug module.

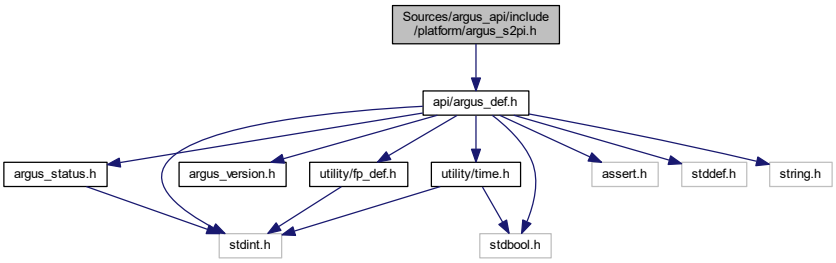
Copyright

11.18 Sources/argus_api/include/platform/argus_s2pi.h File Reference

This file is part of the AFBR-S50 API.

```
#include "api/argus_def.h"
```

Include dependency graph for argus_s2pi.h:



Typedefs

- typedef status_t(∗ s2pi_callback_t) (status_t status, void ∗param)

  S2PI layer callback function type for the SPI transfer completed event.
- typedef void(∗ s2pi_irq_callback_t) (void ∗param)

  S2PI layer callback function type for the GPIO interrupt event.
- typedef int32_t s2pi_slave_t

Enumerations

- enum s2pi_pin_t {
  S2PI_CLK,
  S2PI_CS,
  S2PI_MOSI,
  S2PI_MISO,
  S2PI_IRQ }

Functions

- status_t S2PI_GetStatus (void)

  Returns the status of the SPI module.
- status_t S2PI_TransferFrame (s2pi_slave_t slave, uint8_t const ∗txData, uint8_t ∗rxData, size_t frameSize, s2pi_callback_t callback, void ∗callbackData)

  Transfers a single SPI frame asynchronously.
- status_t S2PI_Abort (void)

  Terminates a currently ongoing asynchronous SPI transfer.
- status_t S2PI_SetIrqCallback (s2pi_slave_t slave, s2pi_irq_callback_t callback, void ∗callbackData)

  Set a callback for the GPIO IRQ for a specified S2PI slave.
- uint32_t S2PI_ReadIrqPin (s2pi_slave_t slave)

  Reads the current status of the IRQ pin.
- status_t S2PI_CycleCsPin (s2pi_slave_t slave)

  Cycles the chip select line.
- status_t S2PI_CaptureGpioControl (void)

  Captures the S2PI pins for GPIO usage.
- status_t S2PI_ReleaseGpioControl (void)

  Releases the S2PI pins from GPIO usage and switches back to SPI mode.
- status_t S2PI_WriteGpioPin (s2pi_slave_t slave, s2pi_pin_t pin, uint32_t value)

  Writes the output for a specified SPI pin in GPIO mode.
- status_t S2PI_ReadGpioPin (s2pi_slave_t slave, s2pi_pin_t pin, uint32_t ∗value)

  Reads the input from a specified SPI pin in GPIO mode.

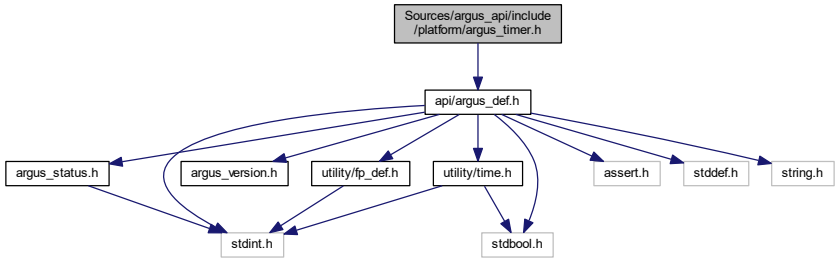
11.18.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides an interface for the required S2PI module.

Copyright

11.19 Sources/argus_api/include/platform/argus_timer.h File Reference

This file is part of the AFBR-S50 API.
`#include "api/argus_def.h"`
Include dependency graph for argus_timer.h:



Typedefs

- typedef void(∗ timer_cb_t) (void ∗param)

  The callback function type for periodic interrupt timer.

Functions

- void Timer_GetCounterValue (uint32_t ∗hct, uint32_t ∗lct)

  Obtains the lifetime counter value from the timers.
- status_t Timer_SetCallback (timer_cb_t f)

  Installs an periodic timer callback function.
- status_t Timer_SetInterval (uint32_t dt_microseconds, void ∗param)

  Sets the timer interval for a specified callback parameter.
- status_t Timer_Start (uint32_t dt_microseconds, void ∗param)

  Starts the timer for a specified callback parameter.
- status_t Timer_Stop (void ∗param)

  Stops the timer for a specified callback parameter.

11.19.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides an interface for the required timer modules.

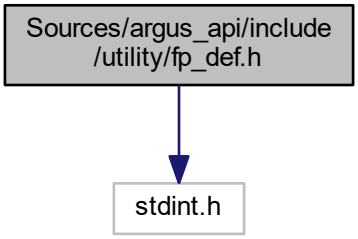
Copyright

11.20 Sources/argus_api/include/utility/fp_def.h File Reference

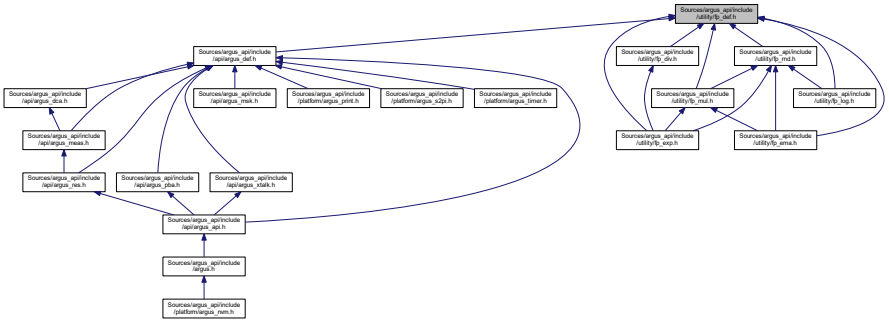
This file is part of the AFBR-S50 API.

```
#include <stdint.h>
```

Include dependency graph for fp_def.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define UQ4_4_MAX ((uq4_4_t)UINT8_MAX)
- #define UQ4_4_ONE ((uq4_4_t)(1U<<4U))
- #define UQ2_6_ONE ((uq2_6_t)(1U<<6U))
- #define UQ0_8_MAX ((uq0_8_t)UINT8_MAX)
- #define UQ12_4_MAX ((uq12_4_t)UINT16_MAX)
- #define UQ12_4_ONE ((uq12_4_t)(1U<<4U))
- #define UQ11_4_ONE ((q11_4_t)(1 << 4))
- #define Q11_4_MAX ((q11_4_t)INT16_MAX)
- #define Q11_4_MIN ((q11_4_t)INT16_MIN)
- #define UQ10_6_MAX ((uq10_6_t)UINT16_MAX)
- #define UQ10_6_ONE ((uq10_6_t)(1U << 6U))
- #define UQ1_15_MAX ((uq1_15_t)UINT16_MAX)
- #define UQ1_15_ONE ((uq1_15_t)(1U << 15U))
- #define UQ0_16_MAX ((uq0_16_t)UINT16_MAX)
- #define UQ0_16_ONE ((uq0_16_t)(1U<<16U))
- #define UQ28_4_MAX ((uq28_4_t)UINT32_MAX)
- #define UQ28_4_ONE ((uq28_4_t)(1U<<4U))
- #define UQ27_4_ONE ((q27_4_t)(1 << 4))

- #define Q27_4_MAX ((q27_4_t)INT32_MAX)
- #define Q27_4_MIN ((q27_4_t)INT32_MIN)
- #define UQ16_16_ONE ((uq16_16_t)(1U << 16U))
- #define UQ16_16_MAX ((uq16_16_t)UINT32_MAX)
- #define UQ16_16_E (0x2B7E1U)
- #define Q15_16_ONE ((q15_16_t)(1 << 16))
- #define Q15_16_MAX ((q15_16_t)INT32_MAX)
- #define Q15_16_MIN ((q15_16_t)INT32_MIN)
- #define Q9_22_ONE ((q9_22_t)(1 << 22))
- #define Q9_22_MAX ((q9_22_t)INT32_MAX)
- #define Q9_22_MIN ((q9_22_t)INT32_MIN)

Typedefs

- typedef uint8_t uq6_2_t

  Unsigned fixed point number: UQ6.2.
- typedef uint8_t uq4_4_t

  Unsigned fixed point number: UQ4.4.
- typedef uint8_t uq2_6_t

  Unsigned fixed point number: UQ2.6.
- typedef uint8_t uq1_7_t

  Unsigned fixed point number: UQ1.7.
- typedef uint8_t uq0_8_t

  Signed fixed point number: Q0.7.
- typedef uint16_t uq12_4_t

  Unsigned fixed point number: UQ12.4.
- typedef int16_t q11_4_t

  Signed fixed point number: Q11.4.
- typedef uint16_t uq10_6_t

  Unsigned fixed point number: UQ10.6.
- typedef uint16_t uq1_15_t

  Unsigned fixed point number: UQ1.15.
- typedef int16_t q0_15_t

  Signed fixed point number: Q0.15.
- typedef int16_t q3_12_t

  Signed fixed point number: Q2.13.
- typedef uint16_t uq0_16_t

  Unsigned fixed point number: UQ0.16.
- typedef uint32_t uq28_4_t

  Unsigned fixed point number: UQ28.4.
- typedef int32_t q27_4_t

  Signed fixed point number: Q27.4.
- typedef uint32_t uq16_16_t

  Unsigned fixed point number: UQ16.16.
- typedef int32_t q15_16_t

  Signed fixed point number: Q15.16.
- typedef uint32_t uq10_22_t

  Unsigned fixed point number: UQ10.22.
- typedef int32_t q9_22_t

  Signed fixed point number: Q9.22.

11.20.1 Detailed Description

This file is part of the AFBR-S50 API.
Provides definitions and basic macros for fixed point data types.

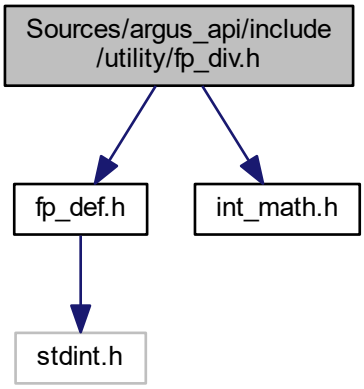
Copyright

11.21 Sources/argus_api/include/utility/fp_div.h File Reference

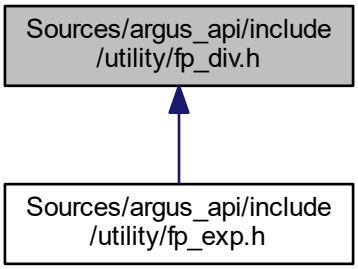
This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include "int_math.h"
```

Include dependency graph for fp_div.h:



This graph shows which files directly or indirectly include this file:



11.21.1 Detailed Description

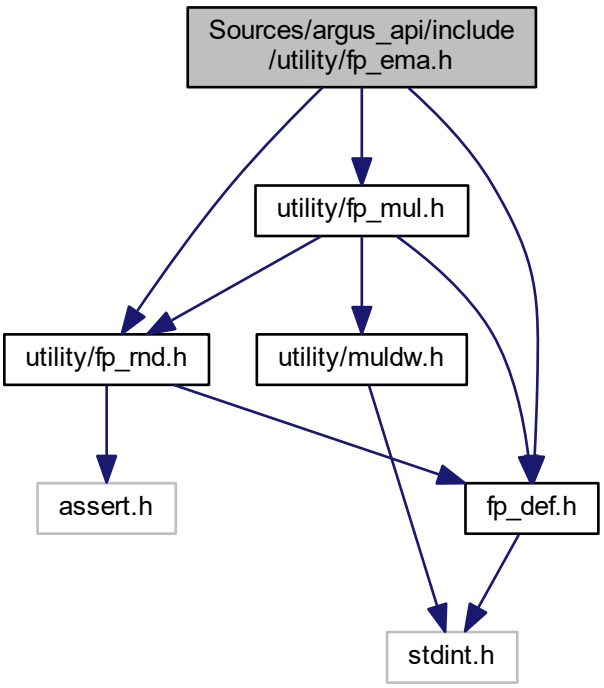
This file is part of the AFBR-S50 API.
Provides definitions and basic macros for fixed point data types.

11.22 Sources/argus_api/include/utility/fp_ema.h File Reference

This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include "utility/fp_rnd.h"
#include "utility/fp_mul.h"
```

Include dependency graph for fp_ema.h:



11.22.1 Detailed Description

This file is part of the AFBR-S50 API.
Provides averaging algorithms for fixed point data types.

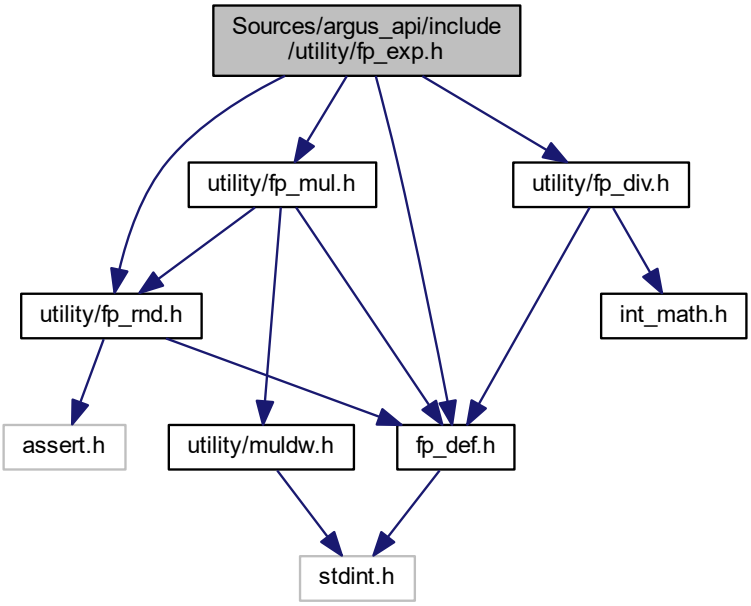
11.23 Sources/argus_api/include/utility/fp_exp.h File Reference

This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include "utility/fp_rnd.h"
#include "utility/fp_mul.h"
```

```
#include "utility/fp_div.h"
```

Include dependency graph for fp_exp.h:



Macros

- #define FP_EXP16_MAX (0x000B1721)
- #define FP_EXP16_MIN (-0x000BC894)
- #define FP_EXP24_MAX (0x058B90BF)

11.23.1 Detailed Description

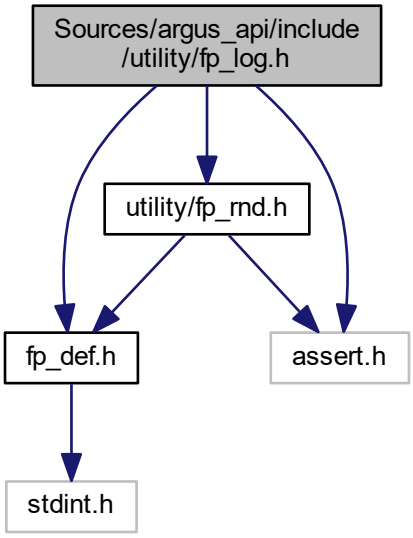
This file is part of the AFBR-S50 API.
This file provides an exponential function for fixed point type.

11.24 Sources/argus_api/include/utility/fp_log.h File Reference

This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include "utility/fp_rnd.h"
#include <assert.h>
```

Include dependency graph for fp_log.h:



Macros

- #define FP_LOG24_2 (0x0B17218)

11.24.1 Detailed Description

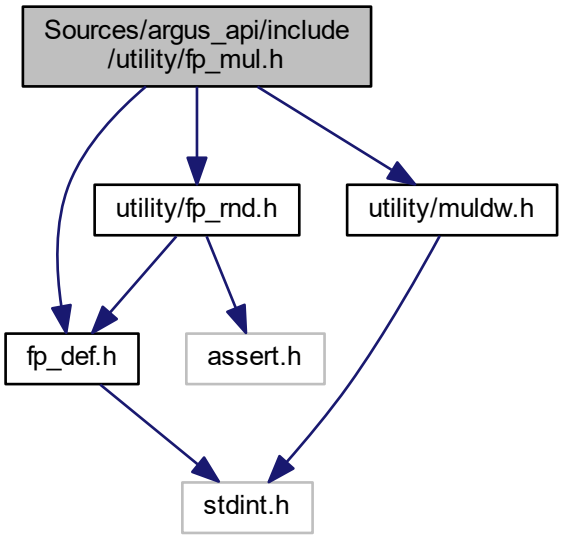
This file is part of the AFBR-S50 API.
This file provides an logarithm function for fixed point type.

11.25 Sources/argus_api/include/utility/fp_mul.h File Reference

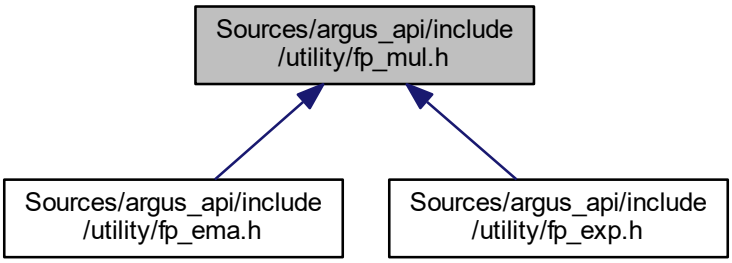
This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include "utility/fp_rnd.h"
#include "utility/muldw.h"
```

Include dependency graph for fp_mul.h:



This graph shows which files directly or indirectly include this file:



11.25.1 Detailed Description

This file is part of the AFBR-S50 API.
Provides definitions and basic macros for fixed point data types.

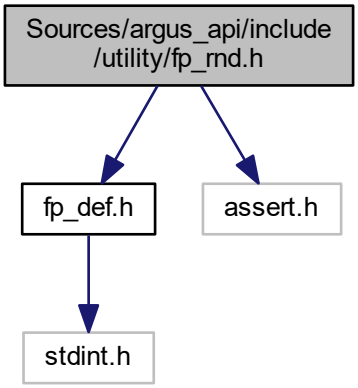
Copyright

11.26 Sources/argus_api/include/utility/fp_rnd.h File Reference

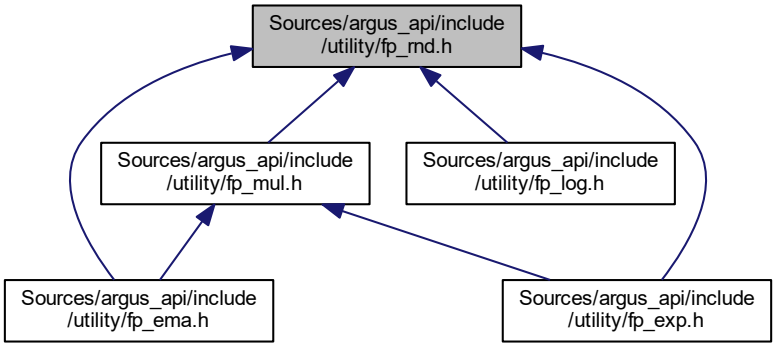
This file is part of the AFBR-S50 API.

```
#include "fp_def.h"
#include <assert.h>
```

Include dependency graph for fp_rnd.h:



This graph shows which files directly or indirectly include this file:



11.26.1 Detailed Description

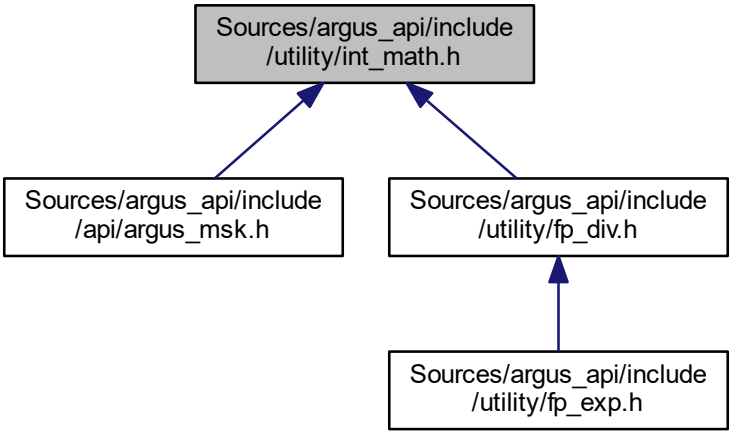
This file is part of the AFBR-S50 API.
Provides definitions and basic macros for fixed point data types.

Copyright

11.27 Sources/argus_api/include/utility/int_math.h File Reference

This file is part of the AFBR-S50 API.

This graph shows which files directly or indirectly include this file:



Macros

- #define INT_SQRT 0
- #define MAX(a, b) ((a) > (b) ? (a) : (b))

  Calculates the maximum of two values.
- #define MIN(a, b) ((a) < (b) ? (a) : (b))

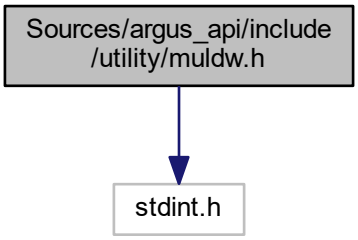
  Calculates the minimum of two values.

11.27.1 Detailed Description

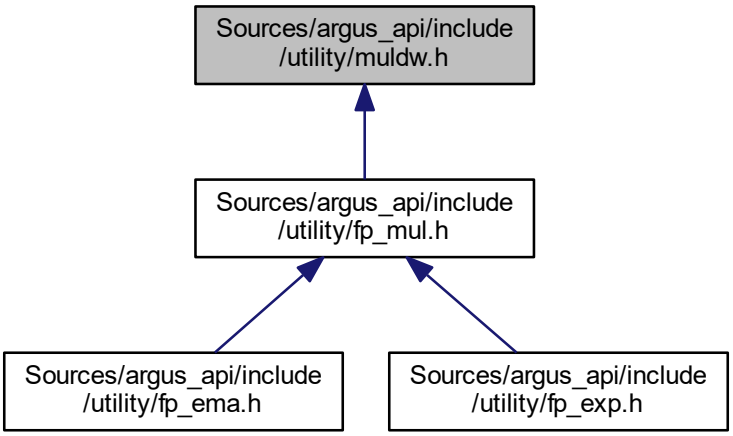
This file is part of the AFBR-S50 API.
Provides algorithms applied to integer values.

11.28 Sources/argus_api/include/utility/muldw.h File Reference

This file is part of the AFBR-S50 API.
`#include <stdint.h>`
Include dependency graph for muldw.h:



This graph shows which files directly or indirectly include this file:

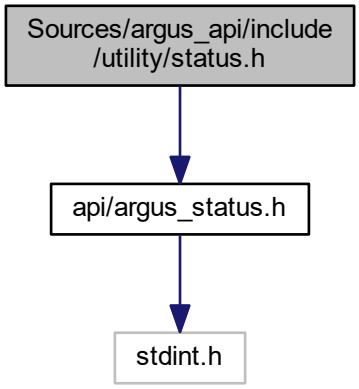


11.28.1 Detailed Description

This file is part of the AFBR-S50 API.
Provides algorithms for multiplying long data types (uint32_t x uint32_t) on a 32-bit architecture.

11.29 Sources/argus_api/include/utility/status.h File Reference

This file is part of the AFBR-S50 API.
`#include "api/argus_status.h"`
Include dependency graph for status.h:



11.29.1 Detailed Description

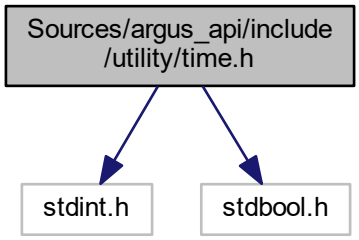
This file is part of the AFBR-S50 API.
This file contains status codes for all platform specific functions.

Copyright

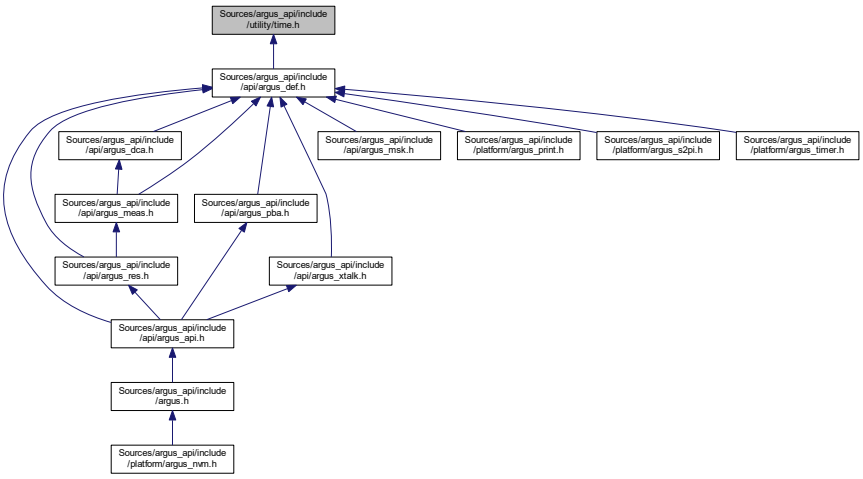
Copyright c 2016-2019, Avago Technologies GmbH. All rights reserved.

11.30 Sources/argus_api/include/utility/time.h File Reference

This file is part of the AFBR-S50 API.
`#include <stdint.h>`
`#include <stdbool.h>`
Include dependency graph for time.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct ltc_t

  A data structure to represent current time.

Functions

- void Time_GetNow (ltc_t *t_now)

  Obtains the elapsed time since MCU startup.
- uint32_t Time_GetNowUSec (void)

  Obtains the elapsed microseconds since MCU startup.
- uint32_t Time_GetNowMSec (void)

  Obtains the elapsed milliseconds since MCU startup.
- uint32_t Time_GetNowSec (void)

  Obtains the elapsed seconds since MCU startup.
- void Time_GetElapsed (ltc_t *t_elapsed, ltc_t *const t_start)

  Obtains the elapsed time since a given time point.
- uint32_t Time_GetElapsedUSec (ltc_t *const t_start)

  Obtains the elapsed microseconds since a given time point.
- uint32_t Time_GetElapsedMSec (ltc_t *const t_start)

  Obtains the elapsed milliseconds since a given time point.
- uint32_t Time_GetElapsedSec (ltc_t *const t_start)

  Obtains the elapsed seconds since a given time point.
- void Time_Diff (ltc_t *t_diff, ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points.
- uint32_t Time_DiffUSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in microseconds.
- uint32_t Time_DiffMSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in milliseconds.
- uint32_t Time_DiffSec (ltc_t const *t_start, ltc_t const *t_end)

  Obtains the time difference between two given time points in seconds.
- void Time_Delay (ltc_t const *dt)

  Time delay for a given time period.

- void Time_DelayUSec (uint32_t dt_usec)

  Time delay for a given time period in microseconds.
- void Time_DelayMSec (uint32_t dt_msec)

  Time delay for a given time period in milliseconds.
- void Time_DelaySec (uint32_t dt_sec)

  Time delay for a given time period in seconds.
- bool Time_CheckTimeout (ltc_t const *t_start, ltc_t const *t_timeout)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutUSec (ltc_t const *t_start, uint32_t const t_timeout_usec)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutMSec (ltc_t const *t_start, uint32_t const t_timeout_msec)

  Checks if timeout is reached from a given starting time.
- bool Time_CheckTimeoutSec (ltc_t const *t_start, uint32_t const t_timeout_sec)

  Checks if timeout is reached from a given starting time.
- void Time_Add (ltc_t *t, ltc_t const *t1, ltc_t const *t2)

  Adds two ltc_t values.
- void Time_AddUSec (ltc_t *t, ltc_t const *t1, uint32_t t2_usec)

  Adds a given time in microseconds to an ltc_t value.
- void Time_AddMSec (ltc_t *t, ltc_t const *t1, uint32_t t2_msec)

  Adds a given time in milliseconds to an ltc_t value.
- void Time_AddSec (ltc_t *t, ltc_t const *t1, uint32_t t2_sec)

  Adds a given time in seconds to an ltc_t value.
- uint32_t Time_ToUSec (ltc_t const *t)

  Converts ltc_t to microseconds (uint32_t).
- uint32_t Time_ToMSec (ltc_t const *t)

  Converts ltc_t to milliseconds (uint32_t).
- uint32_t Time_ToSec (ltc_t const *t)

  Converts ltc_t to seconds (uint32_t).

11.30.1 Detailed Description

This file is part of the AFBR-S50 API.
This file provides utility functions for timing necessities.

Copyright

Chapter 12

Example Documentation

12.1 01_simple_example.c

```
/*******************************************************************************
 * Include Files
 ******************************************************************************/
#include "argus.h"
#include "board/clock_config.h"
#include "driver/cop.h"
#include "driver/gpio.h"
#include "driver/s2pi.h"
#include "driver/uart.h"
#include "driver/timer.h"
/*******************************************************************************
 * Defines
 ******************************************************************************/
#define SPI_SLAVE 1
#define SPI_BAUD_RATE 6000000
/*******************************************************************************
 * Variables
 ******************************************************************************/
static volatile void * myData = 0;
/*******************************************************************************
 * Prototypes
 ******************************************************************************/
/*!*****************************************************************************
 * @brief   printf-like function to send print messages via UART.
 *
 * @details Defined in "driver/uart.c" source file.
 *
 *          Open an UART connection with 115200 bps, 8N1, no handshake to
 *          receive the data on a computer.
 *
 * @param   fmt_s : The usual printf parameters.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
extern status_t print(const char  *fmt_s, ...);
/*!*****************************************************************************
 * @brief   Initialization routine for board hardware and peripherals.
 *
 * @return  -
 ******************************************************************************/
static void hardware_init(void);
/*!*****************************************************************************
 * @brief   Measurement data ready callback function.
 *
 * @details
 *
 * @param   status :
 *
 * @param   data :
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t measurement_ready_callback(status_t status, void * data);
/*******************************************************************************
 * Code
 ******************************************************************************/
/*!*****************************************************************************
 * @brief   Application entry point.
 *
 * @details The main function of the program, called after startup code
 *          This function should never be exited.
 *
```

```
 * @return  -
 ******************************************************************************/
int main(void)
{
    /* The API module handle that contains all data definitions that is
     * required within the API module for the corresponding hardware device.
     * Every call to an API function requires the passing of a pointer to this
     * data structure. */
    argus_hnd_t * hnd = Argus_CreateHandle();
    if (hnd == 0)
    {
        /* Error Handling ...*/
    }
    /* Initialize the platform hardware including the required peripherals
     * for the API. */
    hardware_init();
    /* Initialize the API with default values.
     * This implicitly calls the initialization functions
     * of the underlying API modules.
     *
     * The second parameter is stored and passed to all function calls
     * to the S2PI module. This piece of information can be utilized in
     * order to determine the addressed SPI slave and enabled the usage
     * of multiple devices on a single SPI peripheral. */
    status_t status = Argus_Init(hnd, SPI_SLAVE);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    /* Adjust some configuration parameters by invoking the dedicated API methods. */
    Argus_SetConfigurationFrameTime(hnd, 100000); // 0.1 second = 10 Hz
    /* The program loop ... */
    for(;;)
    {
        myData = 0;
        /* Triggers a single measurement.
         * Note that due to the laser safety algorithms, the method might refuse
         * to restart a measurement when the appropriate time has not been elapsed
         * right now. The function returns with status #STATUS_ARGUS_POWERLIMIT and
         * the function must be called again later. Use the frame time configuration
         * in order to adjust the timing between two measurement frames. */
        status = Argus_TriggerMeasurement(hnd, measurement_ready_callback);
        if (status == STATUS_ARGUS_POWERLIMIT)
        {
            /* Not ready (due to laser safety) to restart the measurement yet.
             * Come back later. */
            __asm("nop");
        }
        else if (status != STATUS_OK)
        {
            /* Error Handling ...*/
        }
        else
        {
            /* Wait until measurement data is ready. */
            do
            {
                status = Argus_GetStatus(hnd);
                __asm("nop");
            }
            while(status == STATUS_BUSY);
            if (status != STATUS_OK)
            {
                /* Error Handling ...*/
            }
            else
            {
                /* The measurement data structure. */
                argus_results_t res;
                /* Evaluate the raw measurement results. */
                status = Argus_EvaluateData(hnd, &res, (void*)myData);
                if (status != STATUS_OK)
                {
                    /* Error Handling ...*/
                }
                else
                {
                    /* Use the recent measurement results
                     * (converting the Q9.22 value to float and print or display it). */
                    print("Range: %d mm\n", res.Bin.Range / (Q9_22_ONE / 1000));
                }
            }
        }
    }
}
static void hardware_init(void)
{
```

```
    /* Initialize the board with clocks. */
    BOARD_ClockInit();
    /* Disable the watchdog timer. */
    COP_Disable();
    /* Init GPIO ports. */
    GPIO_Init();
    /* Initialize timer required by the API. */
    Timer_Init();
    /* Initialize UART for print functionality. */
    UART_Init();
    /* Initialize the S2PI hardware required by the API. */
    S2PI_Init(SPI_SLAVE, SPI_BAUD_RATE);
}
status_t measurement_ready_callback(status_t status, void * data)
{
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    else
    {
        /* Inform the main task about new data ready.
         * Note: do not call the evaluate measurement method
         * from within this callback since it is invoked in
         * a interrupt service routine and should return as
         * soon as possible. */
        myData = data;
    }
    return status;
}
```

12.2 02_advanced_example.c

```
/***************************************************************************
 * Include Files
 **************************************************************************/
#include "argus.h"
#include "board/clock_config.h"
#include "driver/cop.h"
#include "driver/gpio.h"
#include "driver/s2pi.h"
#include "driver/uart.h"
#include "driver/timer.h"
/***************************************************************************
 * Defines
 **************************************************************************/
#define ADVANCED_DEMO 1
#define SPI_SLAVE 1
#define SPI_BAUD_RATE 6000000
/***************************************************************************
 * Variables
 **************************************************************************/
static volatile void * myData = 0;
/***************************************************************************
 * Prototypes
 **************************************************************************/
/*!*************************************************************************
 * @brief   printf-like function to send print messages via UART.
 *
 * @details Defined in "driver/uart.c" source file.
 *
 *          Open an UART connection with 115200 bps, 8N1, no handshake to
 *          receive the data on a computer.
 *
 * @param   fmt_s : The usual printf parameters.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 **************************************************************************/
extern status_t print(const char  *fmt_s, ...);
/*!*************************************************************************
 * @brief   Initialization routine for board hardware and peripherals.
 *
 * @return  -
 **************************************************************************/
static void hardware_init(void);
/*!*************************************************************************
 * @brief   Measurement data ready callback function.
 *
 * @details
 *
 * @param   status :
 *
 * @param   data :
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
```

```
 ******************************************************************************/
status_t measurement_ready_callback(status_t status, void * data);
/*******************************************************************************
 * Code
 ******************************************************************************/
/*!*****************************************************************************
 * @brief   Application entry point.
 *
 * @details The main function of the program, called after startup code
 *          This function should never be exited.
 *
 * @return  -
 ******************************************************************************/
int main(void)
{
    /* The API module handle that contains all data definitions that is
     * required within the API module for the corresponding hardware device.
     * Every call to an API function requires the passing of a pointer to this
     * data structure. */
    argus_hnd_t * hnd = Argus_CreateHandle();
    if (hnd == 0)
    {
        /* Error Handling ...*/
    }
    /* Initialize the platform hardware including the required peripherals
     * for the API. */
    hardware_init();
    /* Initialize the API with default values.
     * This implicitly calls the initialization functions
     * of the underlying API modules.
     *
     * The second parameter is stored and passed to all function calls
     * to the S2PI module. This piece of information can be utilized in
     * order to determine the addressed SPI slave and enabled the usage
     * of multiple devices on a single SPI peripheral. */
    status_t status = Argus_Init(hnd, SPI_SLAVE);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    /* Adjust some configuration parameters by invoking the dedicated API methods. */
    Argus_SetConfigurationFrameTime(hnd, 100000); // 0.1 second = 10 Hz
    /* Start the measurement timers within the API module.
     * The callback is invoked every time a measurement has been finished.
     * The callback is used to schedule the data evaluation routine to the
     * main thread by the user.
     * Note that the timer based measurement is not implemented for multiple
     * instance yet! */
    status = Argus_StartMeasurementTimer(hnd, measurement_ready_callback);
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    for(;;)
    {
        /* Check if new measurement data is ready. */
        if(myData != 0)
        {
            /* Release for next measurement data. */
            void * data = (void *) myData;
            myData = 0;
            /* The measurement data structure. */
            argus_results_t res;
            /* Evaluate the raw measurement results. */
            status = Argus_EvaluateData(hnd, &res, data);
            if (status != STATUS_OK)
            {
                /* Error Handling ...*/
            }
            else
            {
                /* Use the recent measurement results
                 * (converting the Q9.22 value to float and print or display it). */
                print("Range: %d mm\n", res.Bin.Range / (Q9_22_ONE / 1000));
            }
        }
        else
        {
            /* User code here... */
            __asm("nop");
        }
    }
}
static void hardware_init(void)
{
    /* Initialize the board with clocks. */
    BOARD_ClockInit();
```

```
    /* Disable the watchdog timer. */
    COP_Disable();
    /* Init GPIO ports. */
    GPIO_Init();
    /* Initialize timer required by the API. */
    Timer_Init();
    /* Initialize UART for print functionality. */
    UART_Init();
    /* Initialize the S2PI hardware required by the API. */
    S2PI_Init(SPI_SLAVE, SPI_BAUD_RATE);
}
status_t measurement_ready_callback(status_t status, void * data)
{
    if (status != STATUS_OK)
    {
        /* Error Handling ...*/
    }
    else
    {
        /* Inform the main task about new data ready.
         * Note: do not call the evaluate measurement method
         * from within this callback since it is invoked in
         * a interrupt service routine and should return as
         * soon as possible. */
        myData = data;
    }
    return status;
}
```

12.3 sci_python_example.py

```
1 # Example for using the AFBR-S50 API with UART interface
2 # ################################################################################
3 #
4 #
5 # Prepare your evaluation kit (w/ NXP MKL46z MCU) by flashing the UART binary
6 # to the device. Connect the OpenSDA USB port (NOT the one labeled with KL46Z)
7 # to your computer. Go to the Device/Binary folder install directory of you SDK
8 # (default: C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Binary) and copy
9 # the AFBR.S50.ExplorerApp.vX.X.X_KL46z_UART.srec (not the *_USB.*!!) file to
10 # the OpenSDA USB drive.
11 #
12 # After flashing, the device is ready to receive commands via the OpenSDA serial
13 # port. Go to your device manager to find out which COM port is assigned to the
14 # device. Type it to the "port" variable below, before starting the script.
15 #
16 # Use Python 3 to run the script. The script requires the pySerial module which
17 # might need to be installed. See: https://pyserial.readthedocs.io/en/latest/index.html
18 #
19 #
20 # The script sends configuration commands to set the data output mode to 1D data
21 # only and the frame rate to 5 Hz. After setting the configuration, the
22 # measurements are started and the range is extracted from the received data
23 # frames and printed to the console.
24 #
25 #
26 # Note: The CRC values are calculated manually and added before the frames are
27 # sent. You can use the online calculator from the following link w/
28 # CRC8_SAE_J1850_ZERO to obtain the CRC values for a frame:
29 # http://www.sunshine2k.de/coding/javascript/crc/crc_js.html
30 #
31 # ################################################################################
32
33 import serial
34
35 # input parameters
36 port = "COM4"
37 sample_count = 100
38
39
40 # byte stuffing definitions
41 start_byte = b'\x02'
42 stop_byte = b'\x03'
43 esc_byte = b'\x1B'
44
45 def write(tx: bytes):
46
47     print("Sending: " + tx.hex())
48     ser.write(tx)
49
50     return
51
52 def read():
53
54     # read until next stop byte
```

```
55      rx = bytearray(ser.read_until(stop_byte))
56
57      # remove escape bytes if any
58      rxi = rx.split(esc_byte)
59      rx = b''
60      for i in range(len(rxi)):
61          rxi[i][0] ^= 0xFF # invert byte after escape byte (also inverts start byte, but we don't care..)
62      rx = rx.join(rxi)
63
64      # extract command byte (first after start byte)
65      cmd = rx[1]
66
67      # interpret commands
68      if cmd == 0x0A: # Acknowledge
69          print ("Acknowledged Command " + str(rx[2]))
70
71      elif cmd == 0x0B: # Not-Acknowledge
72          print ("Not-Acknowledged Command " + str(rx[2]) + " - Error: " + str((rx[3] << 8) + rx[4]))
73
74      elif cmd == 0x06: # Log Message
75          print("Device Log: " + str(rx[8:-2]))
76
77      elif cmd == 0x36: # 1D Data Set
78          # Extract Range:
79          r = (rx[12] << 16) + (rx[13] << 8) + rx[14]
80          r = r / 16384.0 # convert from Q9.14
81          print ("Range[m]: " + str(r))
82
83      else: # Unknown or not handled here
84          print ("Received Unknown: " + rx.hex())
85
86      return rx
87
88  # open serial port w/ "11500,8,N,1", no timeout
89  print("Open Serial Port " + port)
90  with serial.Serial(port, 115200) as ser:
91      print("Serial Open " + port + ": " + str(ser.is_open))
92
93      # discard old data
94      ser.timeout = 0.1
95      while len(ser.read(100)) > 0: pass
96      ser.timeout = None
97
98      # setting data output mode to 1D data only
99      print("setting data output mode to 1d data only")
100      write(bytes.fromhex('02 41 07 F5 03'))
101      read()
102
103      # setting frame time to 200000 µsec = 0x00030D40 µsec
104      # NOTE: the 0x03 must be escaped and inverted (i.e. use 0x1BFC instead of 0x03)
105      print("setting frame rate to 5 Hz (i.e. frame time to 0.2 sec)")
106      write(bytes.fromhex('02 43 00 1B FC 0D 40 85 03'))
107      read()
108
109      # starting measurements
110      print("starting measurements in timer based auto mode")
111      write(bytes.fromhex('02 11 D0 03'))
112      read()
113
114      #read measurement data
115      print("read measurement data")
116      for i in range(sample_count):
117          read()
118
119      # starting measurements
120      print("stop measurements")
121      write(bytes.fromhex('02 12 F7 03'))
122      read()
123
124      ser.close()             # close port
```