Fraunhofer
IIS

FRAUNHOFER INSTITUTE FOR INTEGRATED CIRCUITS IIS

GETTING STARTED WITH MIOTY^{TM}

Tutorial for setting up and operating a MIOTY^{TM} Network using a Raspberry Pi Pico

GETTING STARTED WITH MIOTY^{TM}

Tutorial for setting up and operating a MIOTY^{TM} Network using a Raspberry Pi Pico

Fraunhofer Institute for Integrated Circuits IIS, Erlangen

Michael Rüger

Contents

1
Raspberry Pi Pico

With the Raspberry Pi Pico, similar to the Arduino, an open-source solution can also be realized. In comparison it is a very fast and powerful microcontroller and therefore also has a slightly higher energy consumption than the other alternatives. It offers a variety of different functionalities that are all well documented. More information about the microcontroller can be found on the official Raspberry Pi Pico website. Of course a variety of sensors can be deployed, but this examples focuses on the use of the commonly used BME temperature, pressure and humidity sensor module. This data is send via the MIOTY^{TM} protocol using an RFM transmitter module.

1.1
Getting Started with the Pico

This chapter focuses on getting started with using the Pico and installing all the required software. At the end of this chapter you are ready to program the Pico and should be able to flash the onboard LED of the microcontroller using an example sketch.
There is a variety of possible programming solutions like C/C++ or MicroPython as a programming language as well as multiple options of development environments. MicroPython is a relatively new programming language that offers fast deployment time. More about this topic can be found on the corresponding website. Probably the most beginner friendly option is the Arduino IDE as a programming solution. It offers good documentation and a wide range of libraries for many different sensors and modules. Unfortunately there is currently no implementation of a dedicated Arduino library for the MIOTY^{TM} protocol using the specific functions of the Pico. The Pico-Foundation offers a collection of libraries and functions, called the PicoSDK, to program the device in C/C++. Documentation of the library can be found here. This tutorial focuses on the deployment of a MIOTY^{TM} sensor node that uses the PicoSDK as a basis of implementation.

1.1.1
Setting up the SDK and blinking the onboard LED

To be able to write and compile code yourself using the functionality of the PicoSDK there are some setup steps that have to be completed first. Depending on the operating system these steps differ, but all achieve the same result. The Raspberry Pi Foundation offers an excellent getting started guide with detailed explanations and pictures of all steps. Generally the steps are based on a Linux-based operating system, but subchapters 9.1 and 9.2 describe the required procedures for setting up the SDK for macOS and Windows respectively. Once the required software is installed and the first program is compiled the different versions can be operated very similar.
The goal of this chapter is to flash the onboard LED of the Pico. Fortunately the Pico Foundation offers a wide variety of examples to demonstrate different functions of the Pico. Hence there is no need to write own code in this step.
To start your development with the Raspberry Pi Pico, follow chapter 2 and chapter 3 for Linux, chapter 9.1 for macOS and 9.2 for Windows machines. After successfully compiling the *pico-examples* project, follow the steps to flash the microcontroller (chapter 3.2) with the *blink.uf2* file. This file should be located in the *pico-examples/build/blink* directory.
If you run into trouble compiling the project, check if the environment variable PICO_SDK_PATH points to the location of the PicoSDK folder. It might help if the complete path is specified, when exporting the variable. Note that this variable has to be

set every time you open a new terminal. Under Linux this command can also be added to the .bashrc-file to have it executed whenever a new terminal is opened:

```
echo “export PICO_SDK_PATH=/path/to/pico-sdk” >> ~/.bashrc
```

It is strongly recommended to use VSCode as a code editor when working with the SDK. It offers additional functionalities when working with CMake as well as an integrated terminal.
Once the steps are successfully completed and the onboard LED of the Pico is flashing, the SDK is setup correctly and you can start developing your own projects.

1.1.2
Hardware Setup

To be able to send data via the MIOTY^{TM} protocol a transmitter module like the HopeRF RFM69W is required. This is also the module used in this tutorial. To communicate between the transmitter and the microcontroller the SPI protocol is used. SPI is a BUS protocol that uses a clock signal to synchronize the data transmission between devices. A select pin determines which device should receive the data. For communication the transmitter requires four wires as well as a supply voltage of 3.3V.

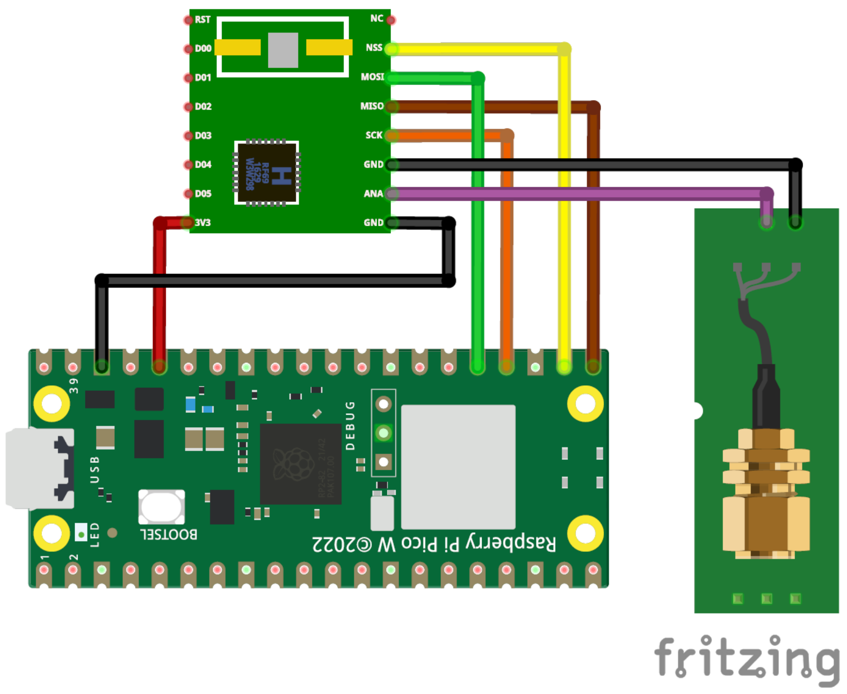
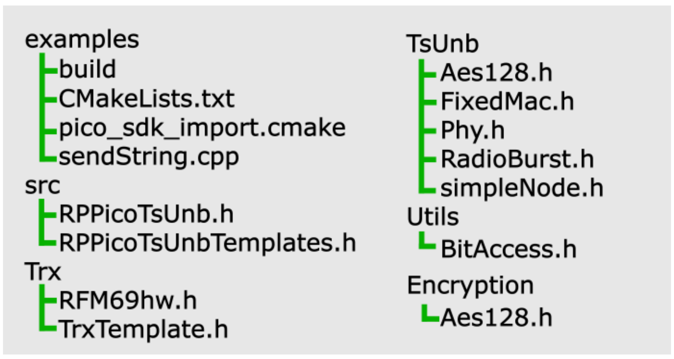


Figure 1 Pico - RFM wiring diagram

To attach an antenna, additionally a SMA connector is connected to ground and the antenna pin of the RFM module. In Figure 1 Pico - RFM wiring diagram shows the minimalistic configuration to be able to send data via the MIOTY^{TM} protocol. It can be expanded to add additional hardware and sensors like a BME temperature sensor. Chapter 1.7 goes into detail what is necessary for the operation of this sensor and how it can be implemented.

1.2
Project Setup

In a previous chapter it is described how to compile and upload example sketches form the RP Pico Examples. If that chapter was completed successfully, further steps can be initiated to setup a project to send the first data via the MIOTY^{TM} protocol. For that a special library was published, called TS-UNB-Lib. It contains a collection of different functions as well as an exemplarily project.
It is recommended to start firstly on the basis of the example project and add additional functionality later. To start, copy all the folders into one directory. Additionally create a new directory called *build* in the *examples* folder. It should have the following structure:



```
examples                      TsUnb
 ┣build                        ┣Aes128.h
 ┣CMakeLists.txt               ┣FixedMac.h
 ┣pico_sdk_import.cmake        ┣Phy.h
 ┗sendString.cpp               ┣RadioBurst.h
src                            ┗simpleNode.h
 ┣RPPicoTsUnb.h               Utils
 ┗RPPicoTsUnbTemplates.h       ┗BitAccess.h
Trx                           Encryption
 ┣RFM69hw.h                    ┗Aes128.h
 ┗TrxTemplate.h
```

Figure 2 - Project Structure

These folders contain header-files that implement different functions for the system. *Txr* is responsible for the sending and receiving functions via the connected RFM69W transceiver module. *TsUnB* implements the MIOTY protocol using telegram splitting technology and *Utils* offers some utility functions. How to use these functions is shown in an example in the *examples* folder. The file *sendString.cpp* initializes the communication with the transceiver module and sends a periodic "Hello World" message via the MIOTY^{TM} protocol.
Once all the files are copied and the path of the PicoSDK is specified, the program can be compiled and run on the Pico.

1.3
Using the TS-Unb-Lib for sending messages via the MIOTY^{TM}

To understand the functions of the library, it is best to demonstrate them using the example sketch that is provided with the library. In summary it firstly initializes the SPI communication to the transceiver module and sets the network key and the EUI of the board. Given these parameters, a periodic *"Hello World"* message is sent via the MIOTY^{TM} protocol. This chapter describes the code in detail and gives some background information.

1.3.1 Pico – Header section

To be able to use all the functions provided by the library, the corresponding header-file is included. It also defines the GPIO-Pins of the Pico used for the SPI communication protocol as well as the baudrate and which of the two SPI interfaces to use. If the RFM module is connected according the wiring diagram in section Hardware Setup there is no need to modify this file. You can check the pinout diagram of the Pico to trace what pins have to be connected if you decide to wire it differently.
Subsequently the network key, EUI and the short address of the device are specified. The network key has a length of 16 byte and defines a unique key used for message assignment and encryption. It should be unique for every board and kept private. The EUI is an individual identifier for every node and used at the base station to distinguish the nodes from another. The short address serves the same purpose as the EUI but is only two bytes in size. It is possible to only transmit the short address instead of the whole EUI to minimize payload size.
The transmit power of the transceiver is specified in dBm. This value is strictly regulated, because MIOTY^{TM} is transmitting in a license-free band, and must not exceed a maximum of 14 dBm. The LED-Pin corresponds to the GPIO Pin connected to the onboard LED of the Pico.

```
#include "../src/RPPicoTsUnb.h"

// This is the node specific configuration
#define MAC_NETWORK_KEY 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88,
0x09, 0xcf, 0x4f, 0x3c
#define MAC_EUI64       0x70, 0xB3, 0xD5, 0x67, 0x70, 0xFF, 0x00, 0x00
#define MAC_SHORT_ADDR  0x70, 0xFF


#define TRANSMIT_PWR    14      // Transmit power in dBm
#define LED_PIN         25

using namespace TsUnbLib::RPPico;

// Select preset depending on TX chip
TsUnb_Rfm69hwEU1_t TsUnb_Node;
```

1.3.2 Pico – Setup section

The *setup()*-function is called once on startup of the microcontroller. It handles all the initialization of the hardware as well as sets the specified parameters of the transmission. In the beginning the LED is initialized and after a short flash the transceiver module is initialized and the parameters are set as well as the packet counter. These functions are pretty self-explanatory. Any further documentation of these functions can be found in the corresponding header file. (Hint: in VSCode hold STRG-Key while clicking on a function to see the implementation). For the correct function of the setup make sure, that this code is executed once at the startup of the microcontroller.

```
//The setup function is called once at startup of the sketch
void setup() {
    sleep_ms(100);

    stdio_init_all();
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    gpio_put(LED_PIN, 1);
    sleep_ms(100);
    gpio_put(LED_PIN, 0);

    TsUnb_Node.init();
    TsUnb_Node.Tx.setTxPower(TRANSMIT_PWR);
    TsUnb_Node.Mac.setNetworkKey(MAC_NETWORK_KEY);
    TsUnb_Node.Mac.setEui64(MAC_EUI64);
    TsUnb_Node.Mac.setShortAddress(MAC_SHORT_ADDR);
    TsUnb_Node.Mac.extPkgCnt = 0x01;

    // Blink LED
    gpio_put(LED_PIN, 1);
    sleep_ms(1000);
    gpio_put(LED_PIN, 0);

}
```

1.3.3 Pico – Main section

This section is executed instantly at the startup of the microcontroller. An important fact is that the *main()*-function must never return. Otherwise the state of the controller is not fixed and the behavior is not deterministic. Therefore an infinite while-loop is required to keep the function from terminating.
Before data can be sent, the initialization process has to be completed. Therefor the *setup()*-function is called once at the beginning. Subsequently data can be sent in periodic intervals by calling the *send()*-method with the following signature:

```
/**
* @brief Send method to transmit a TS-UNB packet
*
* This method transmit the requested payload data. It does the MAC and PHY
* encoding as well as the transmission of the data using the transmitter.
*
* @param payload                 Pointer to payload data
* @param payloadLength           Length of the payload data in bytes
* @param priotry                 Uses low priory uplink pattern if set 6
*
* @return  Non-negative number in case of success, negative number in case of error
*/

int16_t send(const uint8_t* const payload, const uint16_t payloadLength,
             const uint8_t MPF_value = 0, const bool priority= false);
```

The first parameter defines the data that is to be sent via the MIOTY^{TM} protocol. The data has to be split into blocks of 8 bit. Depending on the datatypes of the data that is to be transmitted, a unique representation has to be fulfilled, that is shared between sender and receiver. For example transmitting one uint16_t variable, requires a field of two uint8_t characters. The first one represents the upper bits of the 16 bit variable and the second one the lower part of bits. This can be achieved using bit-shift operation. An example on how this is implemented can be found in section.... .
Furthermore the length of the data field is passed to the function. These two parameters are required for calling the function. To check if the transmission was successful the return value of the method can be verified. In the example code a transmission is indicated by a short flash of the LED.

```
int main() {
    //  wdt_enable();   // Enable 8s supervision TODO:watchdog not implemented
    setup();

    while(1){
        // Send the text "Hello"
        char str[] = "Hello";
        TsUnb_Node.send((uint8_t *)str, sizeof(str) / sizeof(str[0]) - 1);

        // Blink LED when TX is done
        gpio_put(LED_PIN, 1);
        sleep_ms(100);
        gpio_put(LED_PIN, 0);
        sleep_ms(100);
        gpio_put(LED_PIN, 1);
        sleep_ms(100);
        gpio_put(LED_PIN, 0);

        // Sleep some time
        sleep_ms(4000);
    }
}
```

1.4 Compiling and uploading the code

If the project structure complies with the structure given by Figure 2 - Project Structure a *build* folder in the *RPPico* directory should be present. Open a terminal session in the *build* directory (working with the integrated terminal in VSCode is also possible). Firstly the makefile is generated and subsequently the project can be compiled to an executable file.
The commands for building the project differ slightly depending on the operating system. Under Linux based systems run the following commands on the left consecutively and for Windows based machines the ones on the right:

```
cmake ..                    cmake -G "NMake Makefiles" ..
make                        nmake
(Linux)                           (Windows)
```

If errors occur, make sure that the location of the PicoSDK is set correctly. These commands will generate a programmable bit stream for the Pico. After completion several new files should be generated in the *build* folder among others a file called *sendString.uf2*. This executable file is required to program the Pico.
To set the Pico into boot-mode, connect it over USB to the computer while pressing the BOOTSEL-button. The Pico should appear as a storage device. The *.uf2* file simply has to be copied to that device (e.g. drag-and-drop). The execution of the programmed code starts right away. The test message is sent periodically and can be received with a base station.

1.5 Receiving the payload with the base station

The data that is sent by the Pico via MIOTY™ can now be received by a MIOTY™ Gateway. For details on how to register an endpoint to a base station consult the corresponding tutorial.
For the correct data interpretation blueprints are needed to specify the order and size of the transmitted parameters. In the case of the example, the message “Hello” is transmitted. One character has a size of 8 bit. In summary that leads to 40 bit of payload data. The following blueprint can be used for interpreting the transmitted example message.

```
{
    "component": {
        "40bitString": {
            "size": 40,
            "type": "string"
        }
    },
    "meta": {
        "name": "RP Pico",
        "vendor": "Raspberry Pi Foundation"
    },
    "typeEui": "70-b3-d5-67-70-0f-00-b3",
    "uplink": [
        {
            "crypto": 0,
            "id": 0,
            "payload": [
                {
                    "component": "40bitString",
                    "name": "msg"
                }
            ]
        }
    ],
    "version": "1.0"
}
```

1.6 Using print statements in the code

Being able to display outputs of internal states of the microcontroller can be very useful in many cases. Sensor readings can be validated for plausibility and the progress in execution can be checked. Since the Pico is not programmed using an IDE there is no integrated solution to read out the serial output like it is possible with the Arduino IDE. Additional software has to be used to achieve the same result. Also some additions to the *CMakeLists.txt* file have to be made. To enable the output of the serial data via the USB-port, the following line has to be added at the end of the *CMakeLists.txt* file. The name of the executable file, also defined in the same file, has to be specified.

```
pico_enable_stdio_usb(sendString 1)
```

In the source file it might also be necessary to include the standard I/O library via:

```
#include <stdio.h>
```

After these steps are completed, print statements can be used in the code. The output is directed to the USB port during execution. An addition to the given example would be to print that the message has been sent successfully.

```
        // Send the text "Hello"
        char str[] = "Hello";
        TsUnb_Node.send((uint8_t *)str, sizeof(str) / sizeof(str[0]) - 1);
        printf("Message has been transmitted\n");
```

1.6.1 Displaying the output on Linux

To display the data that is sent from the USB port of the microcontroller additional software is required. To read the serial data the program *minicom* is used. The installation process is straight forward. For that open a terminal session and execute the given command.

```
sudo apt install minicom
```

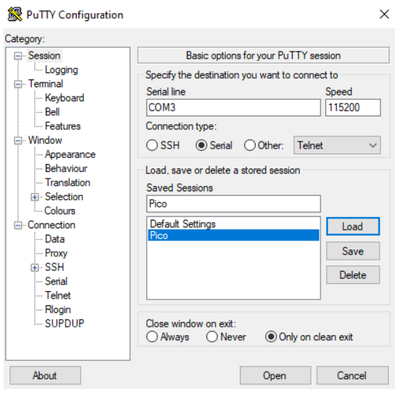
After the installation connect the Pico via USB to the computer. In the terminal a new connection can be initiated with the following command:

```
minicom -b 115200 -o -D /dev/ttyACM0
```

Periodic confirmation messages should be displayed in the window. The serial communication is working as intended. To exit the program press *Ctrl+A* followed by *X*. You can further read about this topic in section 4 of the getting started guide.

1.6.2 Displaying the output on Windows

There are several options to display the serial output of the Pico. A very common and widely used program for the purpose is putty. After the installation connect the Pico via USB to the computer and open the device manager under windows. Under the section *connections (COM & LPT)* should be at least one device listed. If there are multiple devices try re-plugging the Pico and identify the correct port. This port can be used to connect to the device using Putty.



Specify the destination you want to connect to
Serial line: COM3
Speed: 115200
Connection type: SSH [✓ Serial] Other: Telnet
Load, save or delete a stored session
Saved Sessions: Pico
Default Settings
Pico
Close window on exit: Always Never [✓ Only on clean exit]

Figure 3 Putty Configuration

In Putty firstly select *Serial* as the connection type and enter the corresponding COM-port of the Pico. As the communication speed enter a value of 115200. It is also recommended to add this configuration to the saved sessions for faster reconnecting. After opening the session, the messages should appear.

1.7 Integrating a BME Sensor

In the previous chapter first steps were described on how to operate the Pico and send a short test message with help of the MIOTY™ protocol. But for a general use case actual sensor data is transmitted. In this tutorial the steps on how to integrate a sensor are shown using the example of the BME Temperature sensor. In general the Pico offers SPI, I2C and analog input-Pins to communicate with a wide range of sensors and modules. The communication can often be realized with the help of a library but details are dependent on the sensor.
As an example a BME/BMP temperature sensor is chosen to transmit temperature values to the base station. Firstly it is specified how the sensor is wired with the Pico and further the programming of the microcontroller is described. With most BME/BMP sensors there exist multiple ways to communicate with the device. The two communication busses

supported are I2C and SPI. Since the transceiver module RFM is already using the SPI bus, it is suitable to add the BME sensor as a secondary device on the bus.

1.7.1
Wiring of the sensor

Since the Pico offers a variety of SPI capable pins, there are many possible wiring configurations. From a technical point of view the clock, TX and RX pin of the bus are responsible for the synchronous data transmission. To identify which device should receive the data, an additional select signal is required. This means an additional GPIO pin on the microcontroller has to be reserved for each connected device. As soon as data is meant to be sent to a device, the corresponding pin is pulled to ground for the duration of the transmission.
The Pico offers two separate SPI interfaces, which have multiple pins assigned on the board. A detailed description of all the pins and interfaces can be found here. For minimal code changes in the later stages, the sensor module is connected to the default SPI interface on the pins GP16 – GP19. This requires adjustment of the wiring of the transceiver module given in Figure 1 Pico - RFM wiring diagram. The SPI Interface is shared between multiple pins on the microcontroller. For easier wiring the transceiver module is connected on different pins.

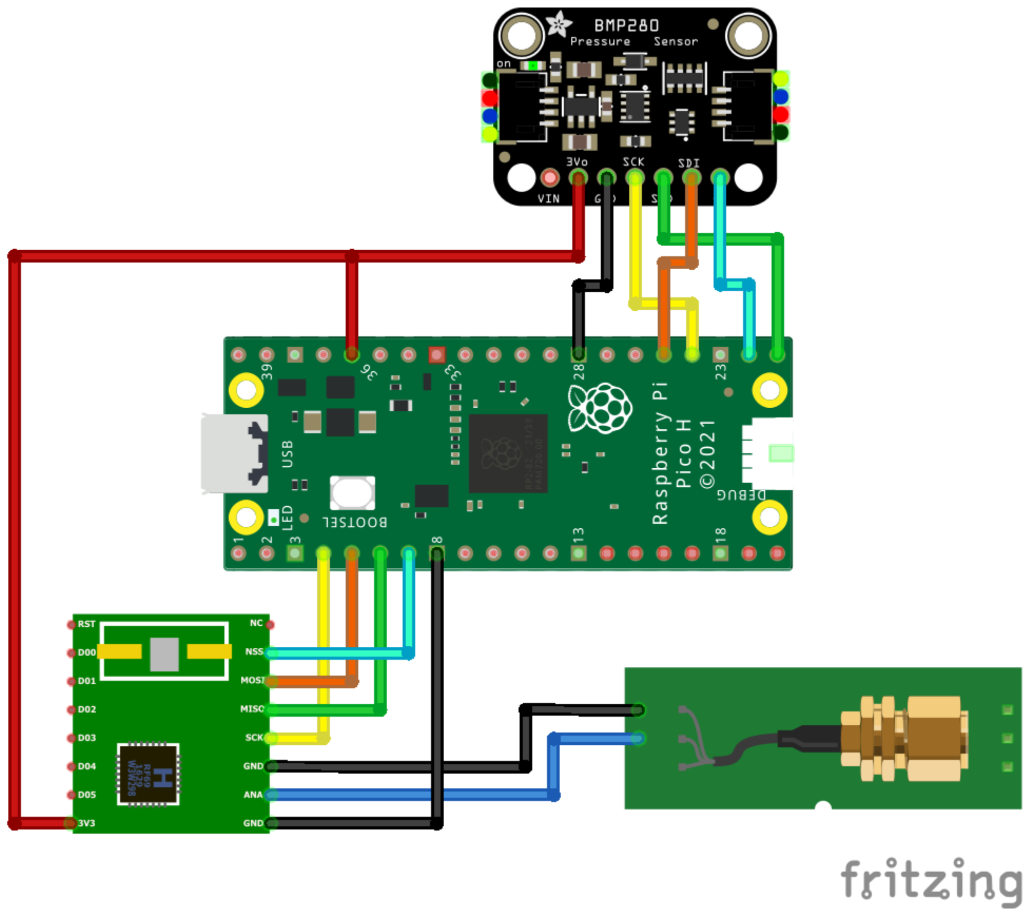


Figure 4 – Wiring Diagram Pico and BME

In the example code in chapter 1.7.2 the hardware is wired according to the diagram in Figure 4 – Wiring Diagram Pico and BME.

1.7.2 Programming the sensor

The Raspberry Pi Foundation offers a variety of different examples on how the Pico and some peripherals can be used. The full list of example sketches, including the source code can be found here. This example on how to transmit temperature data via MIOTY™ is built on the example code for reading out the temperature value of a BME sensor via the SPI interface. It can be found under the SPI section in the list of examples.

To start create a new directory containing all the necessary folders and files for the TS-UNB library, given in chapter Project Setup. Create a new file in the *RPPico* directory called bme_mioty.cpp and copy the code given below into this file (the *sendString.cpp* file is no longer needed and can be deleted). Detail about the MIOTY specific code segments can be found in section Using the TS-Unb-Lib for sending messages via the MIOTYTM.

```
/**
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/spi.h"

// Include Statements for the TsUnb
#include "RPPicoTsUnb.h"


// This is the node specific configuration
#define MAC_NETWORK_KEY 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88,
0x09, 0xcf, 0x4f, 0x3c
#define MAC_EUI64       0x70, 0xB3, 0xD5, 0x67, 0x70, 0xFF, 0x00, 0x00
#define MAC_SHORT_ADDR  0x70, 0xFF


#define TRANSMIT_PWR    14      // Transmit power in dBm
#define LED_PIN         25

using namespace TsUnbLib::RPPico;

// Select preset depending on TX chip
TsUnb_Rfm69hwEU1_t TsUnb_Node;


/* Example code to talk to a bme280 humidity/temperature/pressure sensor.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefore SPI) cannot be used at 5v.

   You will need to use a level shifter on the SPI lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board and a generic bme280 board, other
   boards may vary.

   GPIO 16 (pin 21) MISO/spi0_rx-> SDO/SDO on bme280 board
   GPIO 17 (pin 22) Chip select -> CSB/!CS on bme280 board
   GPIO 18 (pin 24) SCK/spi0_sclk -> SCL/SCK on bme280 board
   GPIO 19 (pin 25) MOSI/spi0_tx -> SDA/SDI on bme280 board
   3.3v (pin 36) -> VCC on bme280 board
   GND (pin 38)  -> GND on bme280 board

   Note: SPI devices can have a number of different naming schemes for pins. See
   the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
   for variations.

   This code uses a bunch of register definitions, and some compensation code derived
   from the Bosch datasheet which can be found here.
   https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf
*/
```

```
#define READ_BIT 0x80

int32_t t_fine;

uint16_t dig_T1;
int16_t dig_T2, dig_T3;
uint16_t dig_P1;
int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
uint8_t dig_H1, dig_H3;
int8_t dig_H6;
int16_t dig_H2, dig_H4, dig_H5;

/* The following compensation functions are required to convert from the raw ADC
data from the chip to something usable. Each chip has a different set of
compensation parameters stored on the chip at point of manufacture, which are
read from the chip at startup and used in these routines.
*/
int32_t compensate_temp(int32_t adc_T) {
    int32_t var1, var2, T;
    var1 = ((((adc_T >> 3) - ((int32_t) dig_T1 << 1))) * ((int32_t) dig_T2)) >> 11;
    var2 = (((((adc_T >> 4) - ((int32_t) dig_T1)) * ((adc_T >> 4) - ((int32_t) dig_T1))) >> 12) *
((int32_t) dig_T3))
            >> 14;

    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}

uint32_t compensate_pressure(int32_t adc_P) {
    int32_t var1, var2;
    uint32_t p;
    var1 = (((int32_t) t_fine) >> 1) - (int32_t) 64000;
    var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((int32_t) dig_P6);
    var2 = var2 + ((var1 * ((int32_t) dig_P5)) << 1);
    var2 = (var2 >> 2) + (((int32_t) dig_P4) << 16);
    var1 = (((dig_P3 * (((var1 >> 2) * (var1 >> 2)) >> 13)) >> 3) + ((((int32_t) dig_P2) * var1)
>> 1)) >> 18;
    var1 = ((((32768 + var1)) * ((int32_t) dig_P1)) >> 15);
    if (var1 == 0)
        return 0;

    p = (((uint32_t) (((int32_t) 1048576) - adc_P) - (var2 >> 12))) * 3125;
    if (p < 0x80000000)
        p = (p << 1) / ((uint32_t) var1);
    else
        p = (p / (uint32_t) var1) * 2;

    var1 = (((int32_t) dig_P9) * ((int32_t) (((p >> 3) * (p >> 3)) >> 13))) >> 12;
    var2 = (((int32_t) (p >> 2)) * ((int32_t) dig_P8)) >> 13;
    p = (uint32_t) ((int32_t) p + ((var1 + var2 + dig_P7) >> 4));

    return p;
}

uint32_t compensate_humidity(int32_t adc_H) {
    int32_t v_x1_u32r;
    v_x1_u32r = (t_fine - ((int32_t) 76800));
    v_x1_u32r = (((((adc_H << 14) - (((int32_t) dig_H4) << 20) - (((int32_t) dig_H5) *
v_x1_u32r)) +
                   ((int32_t) 16384)) >> 15) * (((((((v_x1_u32r * ((int32_t) dig_H6)) >> 10) *
(((v_x1_u32r *

((int32_t) dig_H3))
            >> 11) + ((int32_t) 32768))) >> 10) + ((int32_t) 2097152)) *
                                              ((int32_t) dig_H2) + 8192) >> 14));
    v_x1_u32r = (v_x1_u32r - (((((v_x1_u32r >> 15) * (v_x1_u32r >> 15)) >> 7) * ((int32_t)
dig_H1)) >> 4));
    v_x1_u32r = (v_x1_u32r < 0 ? 0 : v_x1_u32r);
    v_x1_u32r = (v_x1_u32r > 419430400 ? 419430400 : v_x1_u32r);

    return (uint32_t) (v_x1_u32r >> 12);
}

#ifdef PICO_DEFAULT_SPI_CSN_PIN
static inline void cs_select() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 0);  // Active low
    asm volatile("nop \n nop \n nop");
}

static inline void cs_deselect() {
    asm volatile("nop \n nop \n nop");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 1);
    asm volatile("nop \n nop \n nop");
}
#endif

#if defined(spi_default) && defined(PICO_DEFAULT_SPI_CSN_PIN)
static void write_register(uint8_t reg, uint8_t data) {
    uint8_t buf[2];
    buf[0] = reg & 0x7f;  // remove read bit as this is a write
    buf[1] = data;
    cs_select();
    spi_write_blocking(spi_default, buf, 2);
    cs_deselect();
    sleep_ms(10);
}

static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we don't need to keep sending the register we want, just the first.
    reg |= READ_BIT;
    cs_select();
    spi_write_blocking(spi_default, &reg, 1);
    sleep_ms(10);
    spi_read_blocking(spi_default, 0, buf, len);
    cs_deselect();
    sleep_ms(10);
}

/* This function reads the manufacturing assigned compensation parameters from the device */
void read_compensation_parameters() {
    uint8_t buffer[26];

    read_registers(0x88, buffer, 24);

    dig_T1 = buffer[0] | (buffer[1] << 8);
    dig_T2 = buffer[2] | (buffer[3] << 8);
    dig_T3 = buffer[4] | (buffer[5] << 8);

    dig_P1 = buffer[6] | (buffer[7] << 8);
    dig_P2 = buffer[8] | (buffer[9] << 8);
    dig_P3 = buffer[10] | (buffer[11] << 8);
    dig_P4 = buffer[12] | (buffer[13] << 8);
    dig_P5 = buffer[14] | (buffer[15] << 8);
    dig_P6 = buffer[16] | (buffer[17] << 8);
    dig_P7 = buffer[18] | (buffer[19] << 8);
    dig_P8 = buffer[20] | (buffer[21] << 8);
    dig_P9 = buffer[22] | (buffer[23] << 8);

    dig_H1 = buffer[25];

    read_registers(0xE1, buffer, 8);

    dig_H2 = buffer[0] | (buffer[1] << 8);
    dig_H3 = (int8_t) buffer[2];
    dig_H4 = buffer[3] << 4 | (buffer[4] & 0xf);
    dig_H5 = (buffer[5] >> 4) | (buffer[6] << 4);
    dig_H6 = (int8_t) buffer[7];
}

static void bme280_read_raw(int32_t *humidity, int32_t *pressure, int32_t *temperature) {
    uint8_t buffer[8];

    read_registers(0xF7, buffer, 8);
    *pressure = ((uint32_t) buffer[0] << 12) | ((uint32_t) buffer[1] << 4) | (buffer[2] >> 4);
    *temperature = ((uint32_t) buffer[3] << 12) | ((uint32_t) buffer[4] << 4) | (buffer[5] >> 4);
    *humidity = (uint32_t) buffer[6] << 8 | buffer[7];
}
#endif

void bme_setup() {
    #if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) ||
!defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) ||
!defined(PICO_DEFAULT_SPI_CSN_PIN)
#warning spi/bme280_spi example requires a board with SPI pins
    puts("Default SPI pins were not defined");
#else

    printf("Hello, bme280! Reading raw data from registers via SPI...\n");

    // This example will use SPI0 at 0.4MHz.
    spi_init(spi_default, 400000);
    gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
    gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
    // Make the SPI pins available to picotool
    bi_decl(bi_3pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN,
PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI));

    // Chip select is active-low, so we'll initialise it to a driven-high state
    gpio_init(PICO_DEFAULT_SPI_CSN_PIN);
    gpio_set_dir(PICO_DEFAULT_SPI_CSN_PIN, GPIO_OUT);
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 1);
    // Make the CS pin available to picotool
    bi_decl(bi_1pin_with_name(PICO_DEFAULT_SPI_CSN_PIN, "SPI CS"));

    // See if SPI is working - interrograte the device for its I2C ID number, should be 0x60
    uint8_t id;
    read_registers(0xD0, &id, 1);
    printf("Chip ID is 0x%x\n", id);

    read_compensation_parameters();

    write_register(0xF2, 0x1); // Humidity oversampling register - going for x1
    write_register(0xF4, 0x27);// Set rest of oversampling modes and run mode to normal


}
//The setup function is called once at startup of the sketch
void setup() {
    sleep_ms(100);

    stdio_init_all();
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    gpio_put(LED_PIN, 1);
    sleep_ms(100);
    gpio_put(LED_PIN, 0);


    TsUnb_Node.init();
    TsUnb_Node.Tx.setTxPower(TRANSMIT_PWR);
    TsUnb_Node.Mac.setNetworkKey(MAC_NETWORK_KEY);
    TsUnb_Node.Mac.setEui64(MAC_EUI64);
    TsUnb_Node.Mac.setShortAddress(MAC_SHORT_ADDR);
    TsUnb_Node.Mac.extPkgCnt = 0x01;



    // Blink LED
    gpio_put(LED_PIN, 1);
    sleep_ms(1000);
    gpio_put(LED_PIN, 0);

}


int main() {
    stdio_init_all();
    setup();
    bme_setup();

    int32_t humidity, pressure, temperature;
    while (1) {
        spi_init(spi0,400000);
        bme280_read_raw(&humidity, &pressure, &temperature);

        // These are the raw numbers from the chip, so we need to run through the
        // compensations to get human understandable numbers
        pressure = compensate_pressure(pressure);
        temperature = compensate_temp(temperature);
        humidity = compensate_humidity(humidity);

        printf("Humidity = %.2f%%\n", humidity / 1024.0);
        printf("Pressure = %dPa\n", pressure);
        printf("Temp. = %.2fC\n", temperature / 100.0);

        uint16_t payload_transmitter = (int16_t) temperature;
        uint8_t txData [2];
        txData [0] = (payload_transmitter>>8) & 0xFF;
        txData [1] = (payload_transmitter) & 0xFF;
        TsUnb_Node.send(txData,sizeof(txData));
        sleep_ms(4000);
    }

    return 0;
#endif
}
```

1.7.3 Transmitting the payload data

To transmit data over the MIOTY the payload data has to be split into blocks with a size of 8 bit. Hence it is not possible to directly transmit the full range of integer or floating point numbers. It is recommended to figure out the maximum range of possible values and the needed precision of the data first and adjust the size of the data type accordingly. With that the transmission size of the payload is reduced and less bits have to be transmitted.

In the example the temperature value represented by a 16-bit value is transmitted. It offers a range from -32,768 to 32,767. The actual temperature value is obtained by dividing the number by a factor 100. This step is later performed by the base station, when the data is received. The data needs to be packed into an array of 8-bit values before it can be handed to the transmit function as an argument. The first element corresponds to the higher 8 bits of the 16-bit number and the second element to the lower part. Using bit-shift operators this can be achieved. It would be possible to increase the size of the array and therefore transmit additional information. The order and size of the transmitted data has to be known, to be able to decode the data correctly at the base station using blueprints.

```
        uint16_t payload_transmitter = (int16_t) temperature;
        uint8_t txData [2];
        txData [0] = (payload_transmitter>>8) & 0xFF;
        txData [1] = (payload_transmitter) & 0xFF;
        TsUnb_Node.send(txData,sizeof(txData));
```

1.7.4 Adjusting the pins numbers

To account for the correct wiring the pins for the SPI-bus used for communication with the transceiver need to be adjusted. The definition of pins used for communication can be found in *RPPicoTsUnb.h* file.

```
/**
 * @brief Default GPIOs pins for SPI0
 */
#define SPI0_RX      4 // MISO
#define SPI0_CSn     5 // NSS
#define SPI0_SCK     2 // SCK
#define SPI0_TX      3 // MOSI
```

If the wiring of the hardware is done according to section Wiring of the sensor, then the pin configuration listed above can be chosen. Otherwise make sure to connect the SPI capable pins to correct pins on the transceiver module given as a comment in the code section.

1.7.5 CMakeLists for the BME

After the previous steps have been completed the program is ready to be compiled. For that adjustments to the predefined *CMakeLists.txt* file have to be made. Mainly the name of the executable file changed in comparison to the previous example. To be able to display data the output of the print statements, the output over USB is also enabled.

```
cmake_minimum_required(VERSION 3.12)

# Pull in SDK (must be before project)
include(pico_sdk_import.cmake)

project(TS-UNB-Lib C CXX ASM)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

# Initialize the SDK
pico_sdk_init()

add_executable(bme_mioty
        bme_mioty.cpp
        )

# pull in common dependencies and additional spi hardware support
target_link_libraries(bme_mioty pico_stdlib hardware_spi)

# create map/bin/hex file etc.
pico_add_extra_outputs(bme_mioty)

# add url via pico_set_program_url
pico_enable_stdio_usb(bme_mioty 1)
```

The compilation and uploading procedure is similar to the steps described in section Compiling and uploading the code.

1.8 Additional information and material

For exploring different functionalities, that the Pico provides, the Pico examples offer a great first point of contact. Multiple, well documented example sketches are given and can easily be compiled and tested on the hardware.
Of course there are many other different sensor and modules that might need to be integrated and used with the Pico. The SDK Libraries and Tools documentation offers a variety of examples for different sensors and other peripheral devices to easily integrate them into a project. For any other sensor, not included in this list, there are other possibilities for integration. For the majority there already exists an Arduino library that can also be used for the Pico. The article Use Arduino Libraries with the Rasperry Pi Pico C/C++ SDK describes the steps necessary to integrate and use the required libraries. For any other device, there might be the necessity to write an own communication protocol to retrieve the required data. This can be done using the SPI or I2C busses in combination with the functions provided by the PicoSDK.