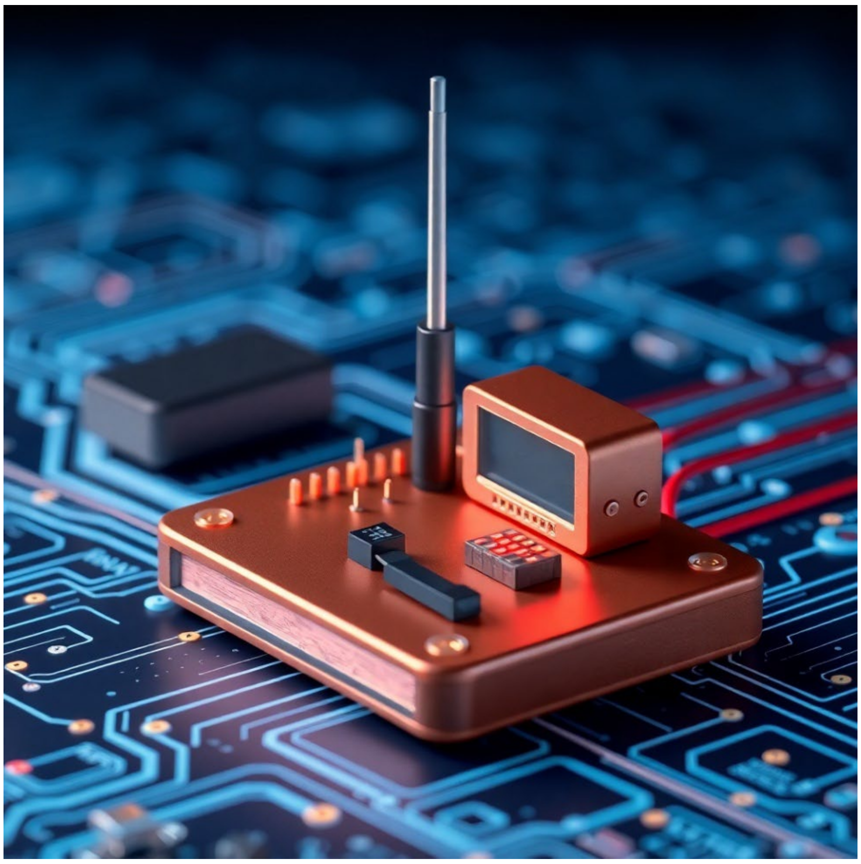


DIY IoT Programming

Content

Introduction

The Internet of Things (IoT) was predicted as far back as 1926 by Nikola Tesla. In an interview, the scientist envisioned a future where all physical objects would connect into a massive system. Today, connected devices are widely used. Some collect environmental data, while others manage processes, finding applications in various industries. You've likely encountered examples of IoT in everyday life, ranging from smart homes to advanced irrigation systems or unmanned aerial vehicles. Many businesses build their operations around creating smart devices or offering IoT-related services.

In this textbook, I will share my experience developing software for such devices, particularly wireless sensors designed to gather information and transmit data wirelessly. Since one of the key criteria for these devices is cost-effectiveness, microcontrollers are used to handle data collection and transmission. This course will teach you how to program microcontrollers, specifically those from the CC13XX family by Texas Instruments. We will use the CC1310 LaunchPad™ development kit, which costs approximately $30 on the manufacturer's website. For data transmission, you will need two such devices—one as a sensor and the other as a receiver. Product link: https://www.ti.com/tool/LAUNCHXL-CC1310.

What you need to know

This tutorial assumes that you have basic knowledge of C programming. In particular, understanding pointer arithmetic, which is a common problem among young programmers.

What will you learn

At the end of the course, you will gain an understanding of how IoT devices are created, and also create your first IoT project. We will create a system from a central device and IoT sensor/s.

Book structure

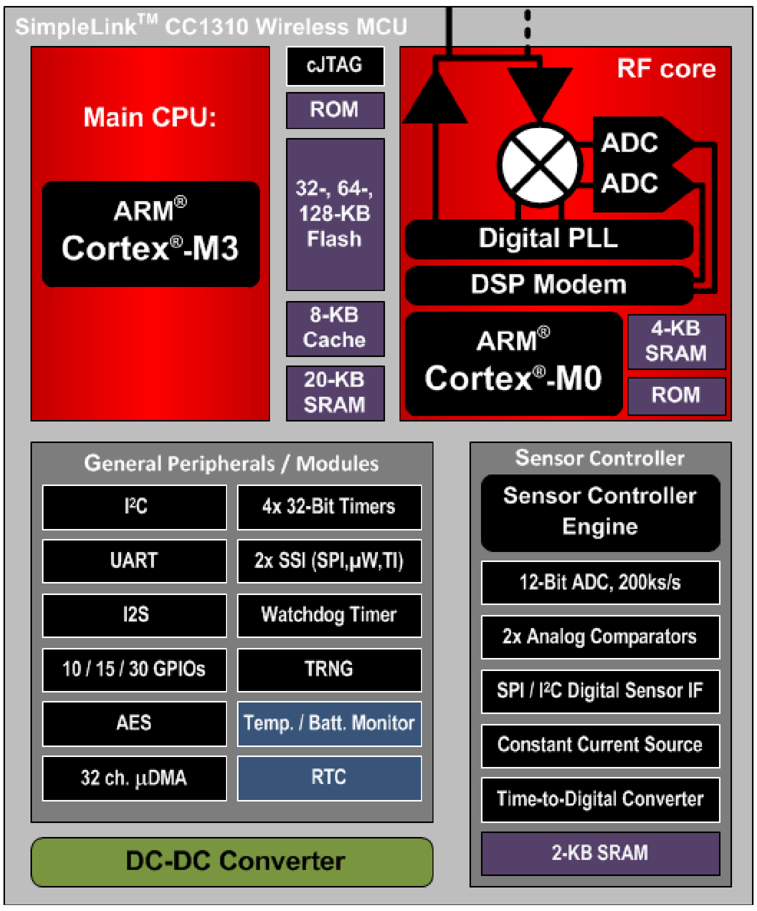
The textbook consists of lessons. The first six lessons describe the operation of the main microcontroller modules that we will use in the final project. At the beginning of the lesson there is a little theory to understand the principle of operation of a particular module. The remaining lessons are the implementation of the project.

Microcontroller

A microcontroller is a chip designed for software control of electronic circuits. It contains both the computing device and the ROM and RAM (as opposed to the microprocessor). In addition, the microcontroller most often contains input/output ports, timers, ADCs, serial and parallel interfaces. An interesting fact is that the first patent for a microcontroller was issued in 1971 to Texas Instruments. One of the main disadvantages of microcontrollers is the amount of memory. Usually it is measured in kilobytes, so we cannot waste much memory and must optimize our code to the maximum in order to fit into the framework allocated to us. On the other hand, microcontrollers are several times cheaper than microprocessors.

So, what does our processor have? If we look at the block diagram, we will see the following modules (Figure 1):

- Main computing device based on Arm® Cortex®-M3 processor. This module apparently executes our application (program)
- ROM contains a basic bootloader and some built-in functions
- Flash – 32kB, 64kB or 128kB (depending on version) memory to store application or data.
- SRAM – memory used by the application (20 kB)
- GPRAM – memory that can be used as a cache or as regular memory (8 kB)
- 4 32-bit timers
- RTC (real-time clock)
- TRNG (random number generator)
- Watchdog
- General Purpose Input-Outputs (GPIO)
- 12-bit ADC
- Encryption module AES-128
- Built-in temperature and voltage sensors
- 32-channel DMA
- Interfaces I2C, UART, I2S, SPI
- Ultra-low power standalone sensor controller (2 kB memory)


SimpleLink™ CC1310 Wireless MCU
Main CPU:
ARM® Cortex®-M3
cJTAG
ROM
32-, 64-, 128-KB Flash
8-KB Cache
20-KB SRAM
RF core
ADC
ADC
Digital PLL
DSP Modem
ARM® Cortex®-M0
4-KB SRAM
ROM
General Peripherals / Modules
I²C
4x 32-Bit Timers
UART
2x SSI (SPI,µW,TI)
I2S
Watchdog Timer
10 / 15 / 30 GPIOs
TRNG
AES
Temp. / Batt. Monitor
32 ch. µDMA
RTC
DC-DC Converter
Sensor Controller
Sensor Controller Engine
12-Bit ADC, 200ks/s
2x Analog Comparators
SPI / I²C Digital Sensor IF
Constant Current Source
Time-to-Digital Converter
2-KB SRAM

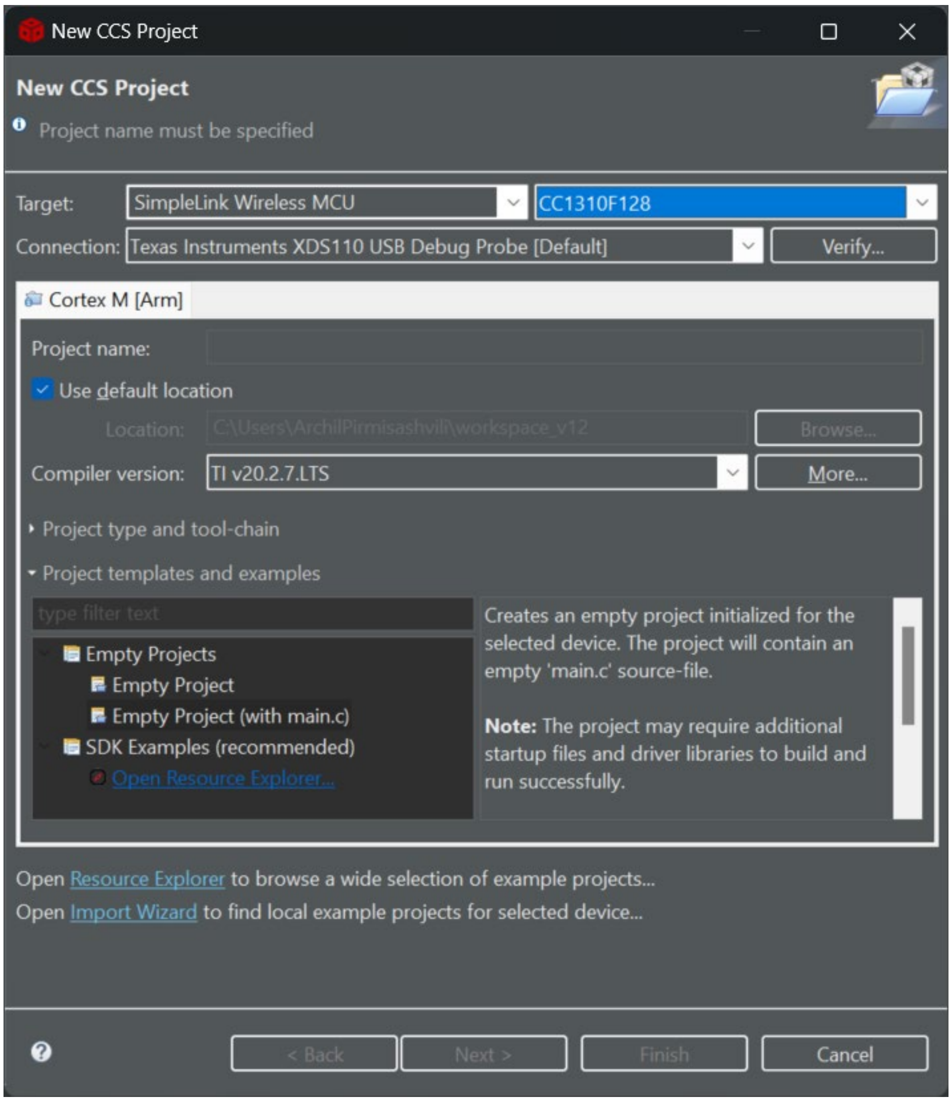
Preparing the working environment

To develop microcontroller software we will need the following tools:

- Code generation tools (compiler) - https://www.ti.com/tool/ARM-CGT
- Software development it (SDK) - https://www.ti.com/tool/SIMPLELINK-CC13X0-SDK
- Integrated Development Environment - https://www.ti.com/tool/CCSTUDIO
- Application for firmware flashing (optional) - https://www.ti.com/tool/FLASH-PROGRAMMER

Creating a Project

So, all the tools are installed and we can't wait to get started. Let's start by creating a project. Launch the development environment (Code Composer Studio IDE) and create our first project.


New CCS Project

New CCS Project

Project name must be specified

Target: SimpleLink Wireless MCU | CC1310F128

Connection: Texas Instruments XDS110 USB Debug Probe [Default]

Cortex M [Arm]

Project name:

Use default location

Location: C:\Users\ArchilPirmisashvili\workspace_v12

Compiler version: TI v20.2.7.LTS

Project type and tool-chain

Project templates and examples

Empty Projects
- Empty Project
- Empty Project (with main.c)

SDK Examples (recommended)
- Open Resource Explorer...

Creates an empty project initialized for the selected device. The project will contain an empty 'main.c' source-file.

Note: The project may require additional startup files and driver libraries to build and run successfully.

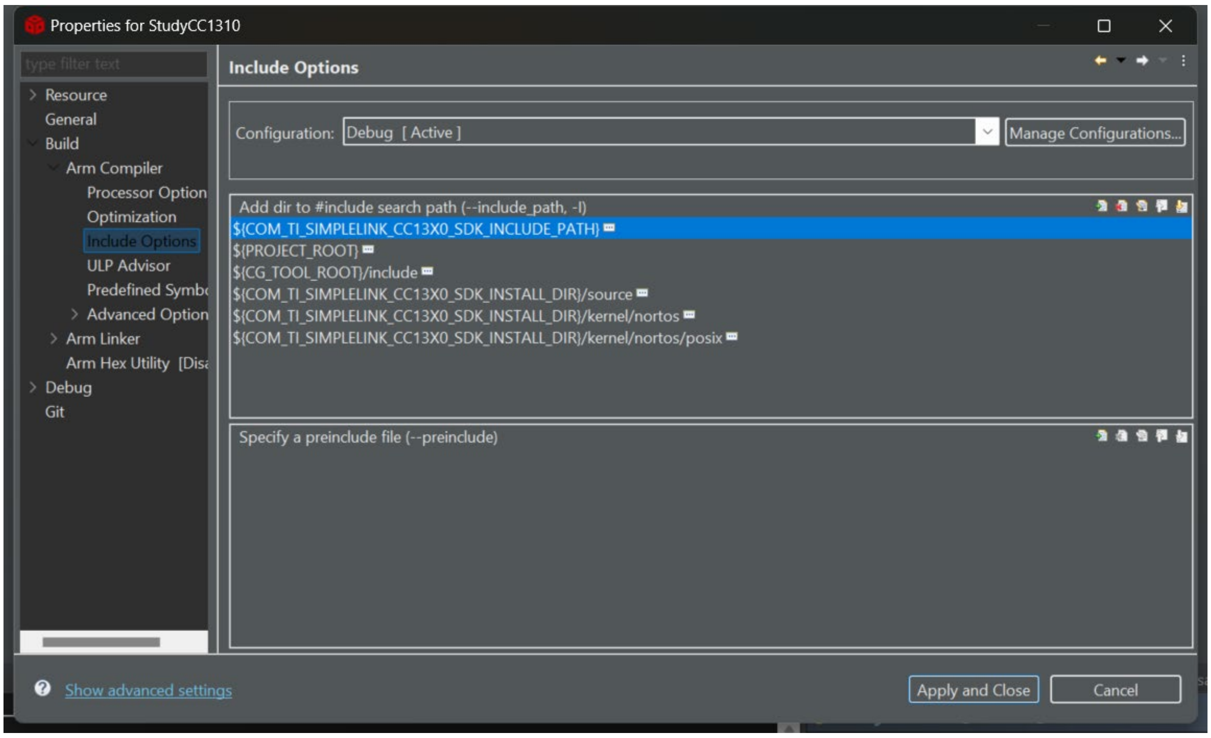
Open Resource Explorer to browse a wide selection of example projects...

Open Import Wizard to find local example projects for selected device...

In the “Target” field, select “Simplelink Wireless MCU”, and the controller model “CC1310F128”. Create a project with the main.c file.

Next, we configure the compiler and linker. To do this, open the Project Properties window (key combination Alt + Enter). In the General department, open Products and add Simplelink CC13x0 SDK. Next, add the paths to the SDK for the compiler (Build->Arm Compiler->Include Options):

```
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/source
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/nortos
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/norstos/posix
```


Properties for StudyCC1310

Include Options

Configuration: Debug [Active]

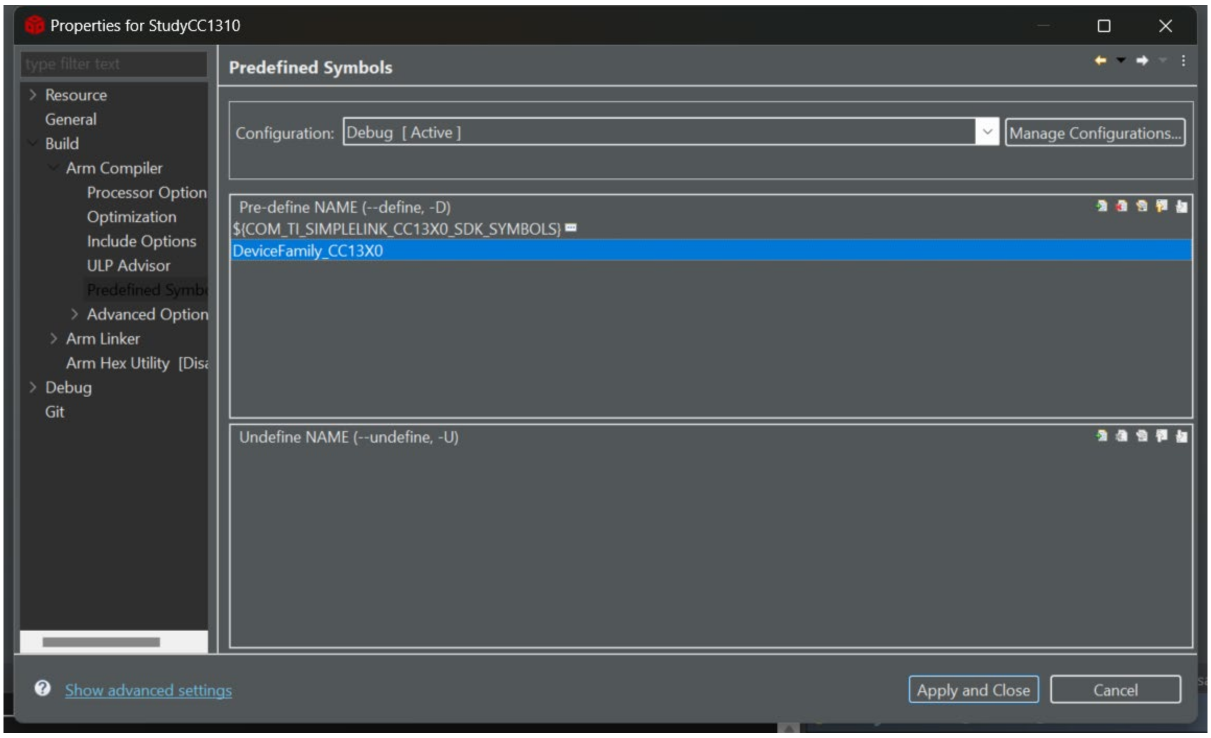
Add dir to #include search path (--include_path, -I)

```
${COM_TI_SIMPLELINK_CC13X0_SDK_INCLUDE_PATH}
${PROJECT_ROOT}
${CG_TOOL_ROOT}/include
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/source
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/nortos
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/nortos/posix
```

Specify a preinclude file (--preinclude)

Next, in Predefined Symbols (Build->Arm Compiler->Predefined Symbols) add:

```
DeviceFamily_CC13X0
```


Properties for StudyCC1310

Predefined Symbols

Configuration: Debug [Active]

Pre-define NAME (--define, -D)

${COM_TI_SIMPLELINK_CC13X0_SDK_SYMBOLS}

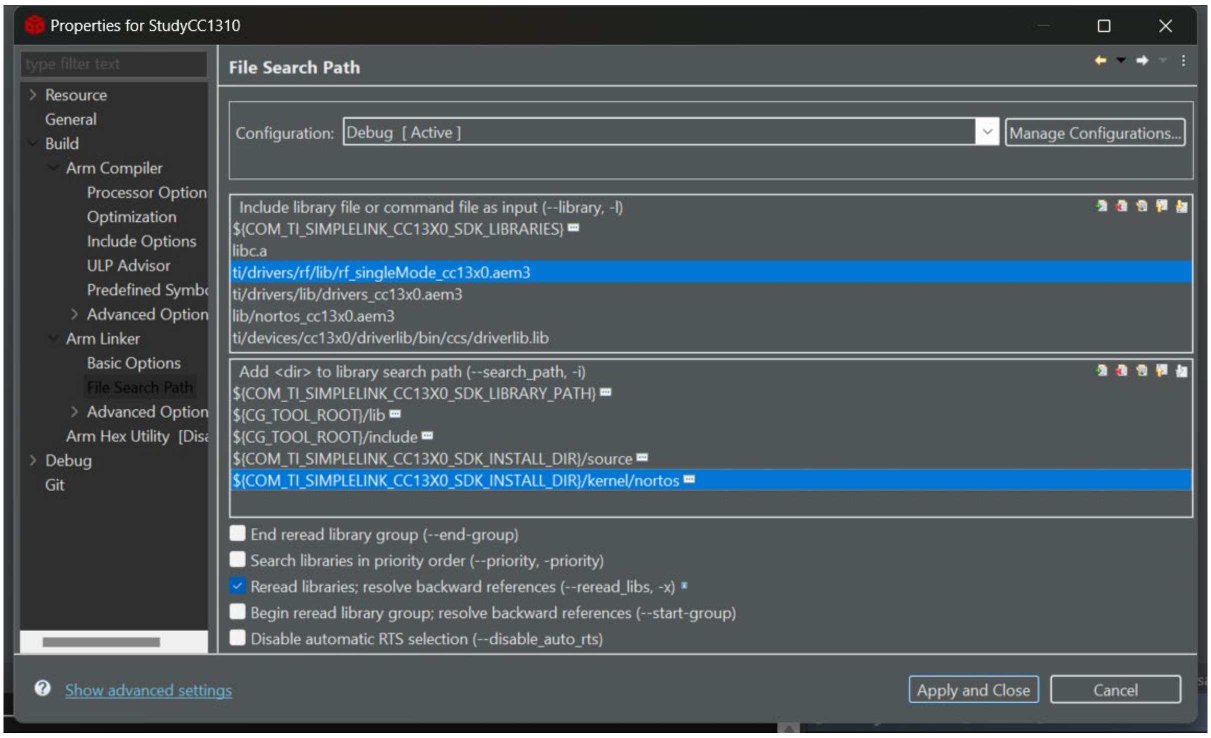
DeviceFamily_CC13X0

Undefine NAME (--undefine, -U)

Setting up the linker (Build->Arm Linker->File Search Path). Add paths to libraries and connect them.

```
ti/drivers/rf/lib/rf_singleMode_cc13x0.aem3
ti/drivers/lib/drivers_cc13x0.aem3
lib/nortos_cc13x0.aem3
ti/devices/cc13x0/driverlib/bin/ccs/driverlib.lib
```

```
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/source
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/nortos
```


Properties for StudyCC1310

File Search Path

Configuration: Debug [Active]

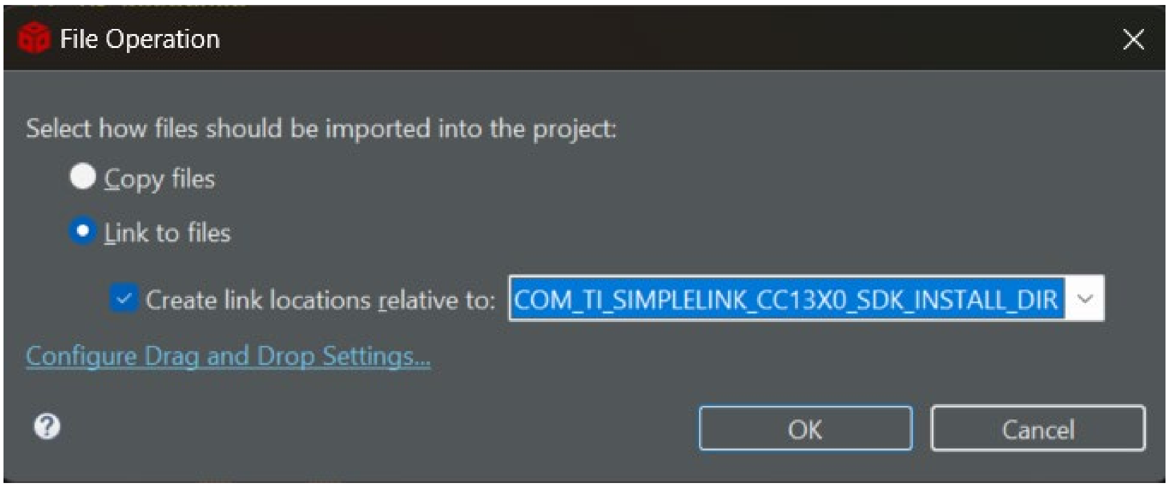
Include library file or command file as input (--library, -l)
```
${COM_TI_SIMPLELINK_CC13X0_SDK_LIBRARIES}
libc.a
ti/drivers/rf/lib/rf_singleMode_cc13x0.aem3
ti/drivers/lib/drivers_cc13x0.aem3
lib/nortos_cc13x0.aem3
ti/devices/cc13x0/driverlib/bin/ccs/driverlib.lib
```

Add <dir> to library search path (--search_path, -i)
```
${COM_TI_SIMPLELINK_CC13X0_SDK_LIBRARY_PATH}
${CG_TOOL_ROOT}/lib
${CG_TOOL_ROOT}/include
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/source
${COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR}/kernel/nortos
```

- [ ] End reread library group (--end-group)
- [ ] Search libraries in priority order (--priority, -priority)
- [✓] Reread libraries; resolve backward references (--reread_libs, -x)
- [ ] Begin reread library group; resolve backward references (--start-group)
- [ ] Disable automatic RTS selection (--disable_auto_rts)

Please note that the project has two configurations Debug and Release. The second configuration optimizes the code and generates a smaller executable file. Therefore, these settings should be done in both configurations.

The creators of the SDK took care of us and prepared a file that contains the microcontroller interrupt vector settings. Let's drag it (startup_ccs.c) into our project and add it as a link (from the folder SDK/source/ti/devices/cc13x0/startup_files).


File Operation

Select how files should be imported into the project:

- Copy files
- [✓ Link to files]

[✓] Create link locations relative to: COM_TI_SIMPLELINK_CC13X0_SDK_INSTALL_DIR

That's it, setup is complete. Now our project can be successfully built.

Features of microcontroller programming

Since microcontrollers have little memory, we must try to save on "everything we can." Therefore, unless absolutely necessary, we should not use all the capabilities of

modern computers, namely, in order to “light a light bulb” or “send a message” or something like that, there is no particular need to use the operating system. Of course, there are operating systems for microcontrollers, but in our case their use is unnecessary.
Another important rule for programming microcontrollers is the prohibition on the use of dynamic memory allocation or the use of the malloc function, which is understandable to every programmer. The fact is that embedded systems have small memory and over time, frequent calls to memory allocation/deallocation functions can cause severe fragmentation and memory leaks. And very often the use of dynamic memory is one of the main causes of bugs.

Many new programmers do not understand the meaning of the volatile keyword. And in embedded systems it is often used. The purpose of this keyword is to prevent the compiler from optimizing the variable. For example, take the following pseudocode:

```
int i = 0;

void irq_callback() {
    i++;
}

void main() {
    while (!i) {
    }
}
```

The problem with this code is that the variable i is not changed in the while loop, but in the interrupt handler, in other words, from a different thread. The compiler will understand that the variable does not change its value and the loop condition is always satisfied, which means it can be optimized into the following code:

```
while(1) {
}
```

Quite often compilers do not optimize such a variable, but if we increase the optimization level, we end up with broken code. Then we spend a lot of time trying to find the cause of the problem. This is why it is important to add the volatile keyword.

Project structure

For ease of use and understanding of the code, we will divide the project into folders. Each folder will contain files with specific parts/modules.

Boards – in this folder we will store files related to a specific device. In this tutorial we will only use one device, but if you decide to add a new one, it will not be too difficult for you.

Drivers – we will store drivers in this folder. Mainly peripheral/interface drivers.

Apps – in this folder we will store application files that we will launch on the device.

Tests – in this folder we will store test files that we will use to test the functionality of our code.

Tools – in this folder we will store auxiliary files such as batch files.

The ti-lib.h file contains auxiliary macros and developer library headers.

SDK

In this tutorial we will use the Simplelink CC13X0 SDK. This library offers us a convenient interface for working with a microcontroller. Essentially, working with any microcontroller consists of working with its registers, changing the values of which we can use the modules/peripherals of the microcontroller. Working with registers is not particularly convenient for the average person; for this purpose, the microcontroller developer offers us a library of functions understandable to the average programmer.

One of the main modules of the library is NoRTOS. This module gives us the ability to work with a microcontroller without an operating system. It allows us to use delays, takes care of interrupt registration, system clock, power saving management and much more.

Source code

You can find the source code at:
https://github.com/Kulipator/DIY-IoT-Programming
Each part is in a separate branch.

Superloop architecture

The architecture of the microcontroller, as well as the microprocessor, is designed in such a way that from the moment power is supplied to the microcontroller, in other words, turned on, it must constantly execute commands. Even when idle, it executes the so-called NOP command. In the case of operating systems, the latter takes over this responsibility, in other words, when there is no program running, the operating system starts the IDLE process (idle process). In our case, since we do not have an operating system, our program should never terminate. In other words, our program should have an infinite loop. Hence the name of the architecture.

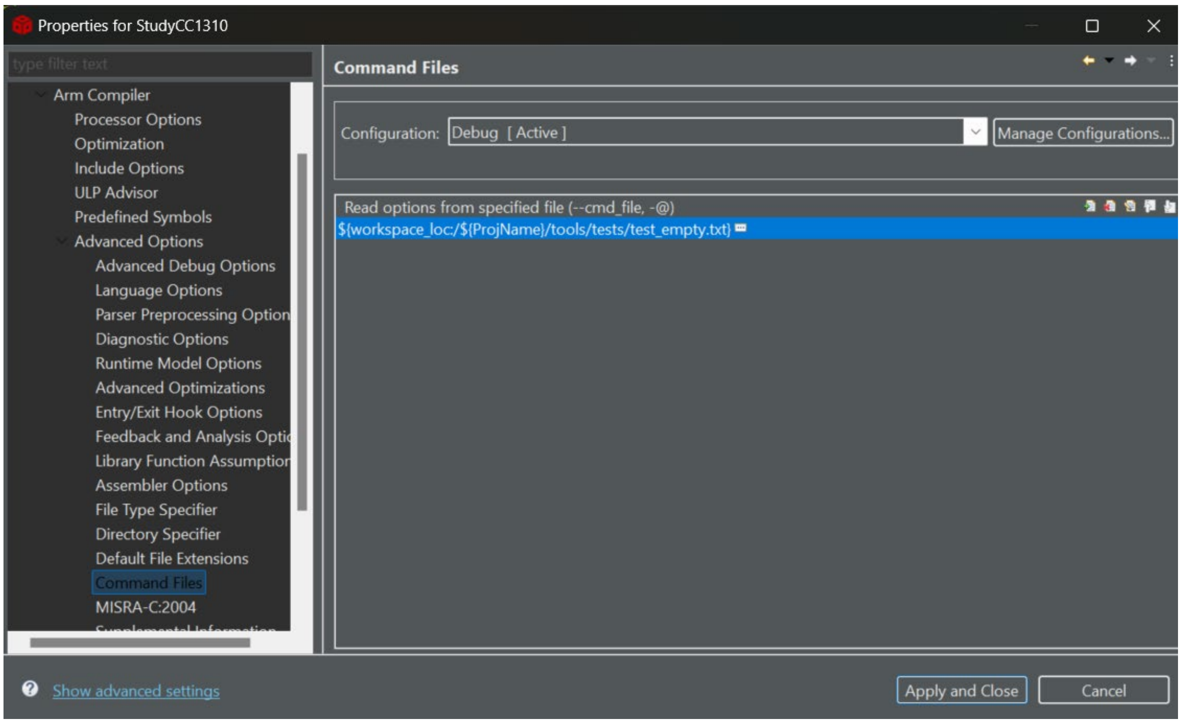
This architecture implies a special programming concept - asynchronous. Let's take an example, we have a system that must process user input and send data over a radio channel and vice versa, receive data from a radio channel and send it to the user for output. The simplest solution to this problem is to wait for user input, and upon receipt of input send data, but what happens if at this moment data arrives from the radio channel? We will not be able to process them because we are busy receiving data from the user, and only after we receive the user input will we be able to process the data from the radio channel. This solution is not suitable for us, as it leads to delays, and in the case of a large amount of data, to buffer overflow and data loss. Therefore, the function for reading user input should not wait for the end of the input, but return immediately, for example, check whether there is data and, if there is any, process it, otherwise exit.

Microcontroller programming

First application

The first application we will create is an empty application that initializes our device and does nothing else. We will add to it as we progress.
The initial entry point is in the main.c file, the main function. Let's see what she does? First of all, we must initialize the peripherals of our device by calling the board_initialize function. Next, we initialize the NoRTOS library module and transfer control to our application by calling its launch function (mainThread). In case we mistakenly exit the main function of the application, we will add an infinite loop that will prevent our program from ending. This cycle plays the role of protection; if we do everything correctly, then we will never reach this cycle.

Since this application doesn't do anything, let's consider it a test. Therefore, let's place the main application file in the Tests folder. If you pay attention, in this file the code is enclosed in the #idef preprocessor directive, this approach allows us to have many files with the mainThread function and use the file we need using the preprocessor directive. We will place this directive in the command file in the tools folder. At the compilation stage, we will need to select the command file we need, and the compiler will build the required application for us. To select a command file, open the project settings and select the desired file in the Build->Arm Compiler->Advanced Options->Command Files section.



Configuration: Debug [Active]

Read options from specified file (--cmd_file, -@)

```
${workspace_loc:/${ProjName}/tools/tests/test_empty.txt}
```

You can see that there are two definitions in the test_empty.txt file: device (BOARD) and application.

The application file empty_test.c essentially has an infinite loop in the main function. Thus the application does nothing.

Now let's look at the boards folder. In it we have the file board_common, which contains functions common to all devices, for example the initialization function, as well as a prototype of the application login function (mainThread).

Also, you can see that depending on the definition of BOARD, the necessary files from the device folder are connected.

Let's try to build our first application. To do this, press the key combination CTRL + B, or click on the icon with a picture of a hammer. And then we can launch the application in debugging mode by clicking the icon with a bug design. Viola, our first program is running on the device, but we don't see any sign of it running since it doesn't do anything.

Let there be light!!!

Our first application did nothing and it was not clear whether it worked or not. Let's add some interaction to it. The easiest way is to light a light bulb (LED), we even have two of them on our device. But first, a little theory and electronics.

We all know that in order to light a light bulb, you need to pass current through it. But how to do this on our device? Essentially, a light bulb should have two states, on or off. To monitor these states, a digital input/output (GPIO) module is ideal for us.

So how does the GPIO module work? This module is one of the main ones in every microcontroller. There are a number of pins that can perform GPIO functions. Each such input/output has a number and can operate in 2 states (either input or output), in other words, take state 0 or 1. But how does the controller know where 1 is and where 0 is. To do this, let's try to understand how digital electronics work. For any device to operate, a power source is required; it powers the electrical circuit through electric current. But how can current turn into digital? Everything is very simple, if there is current, then it is a logical unit, if there is no current, then it is 0. But in fact, not everything is quite like that. The fact is that a logical unit is considered to be a voltage gap close to the upper limit of the power source voltage (in electronics 3.3 volts or 5 volts, depending on the system), and a logical 0 is a voltage gap close to 0. In the case of a light bulb, if we use the digital output of the device and switch its state to 1, then voltage will appear at its output and if we connect a light bulb to it, it will light up.

Let's look at the GPIO modes. As we already understood, there are two modes: input or output. Each of them has its own modes.

Input modes

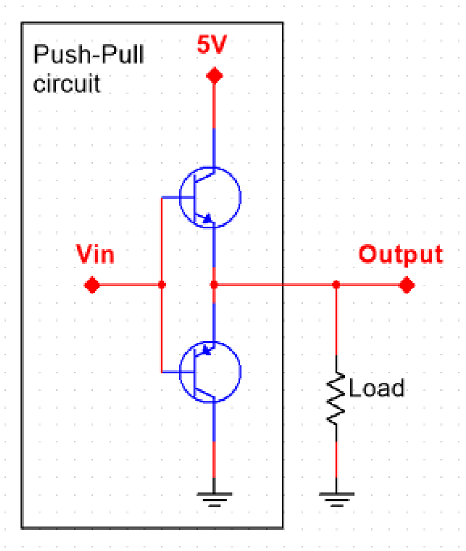
Normal (No pull) – This input mode is used by default in microcontrollers. The input voltage is converted to a logical value and output as a measurement result. But there's a problem. What happens if the input voltage is somewhere between 0 and 1? To solve this situation, we need to "pull" the tension in one direction. This can be solved by using a pull-up resistance. This resistor can be supplied as a physical component, or use the built-in GPIO pull-up mode.

Pull up - this input mode is used when the circuit takes the value 0, or unknown. A classic example is a button/switch. If the circuit is constructed in such a way that the switch connects the input to ground, then when the switch is open, an unknown value (or 0) is generated at the input, in which case we will not be able to understand whether the button is pressed or not. But if we "pull" the voltage up, then in the case of an open switch we will get 1 at the input, and in the case of a closed switch – 0.

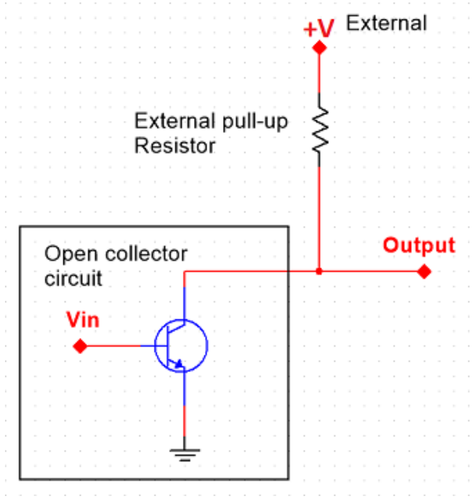
Pull down – the reverse mode from the previous one. In it we "pull" the tension down. As an example, we can use the same switch only with a closing input with +.

Output modes

PushPull - in this mode, the output takes the value either 1 or 0. In other words, it cannot be unknown, since it is either pulled up or down. Hence the name.



OpenDrain - in this mode, the output takes the value either 0 or unknown. Essentially, this mode is used by various data transfer interfaces, such as I2C, when there are several devices on the bus and in order not to interfere with the operation of other devices. To understand why this mode is needed, let's try to connect two PushPull pins and put them in different states. Due to a conflict of levels (potentials), current will flow in the circuit and one of the outputs may burn out.



OpenSource - this mode is similar to the previous one with the difference that instead of 0 it receives the value 1.

We've sorted out the theory, let's move on to practice. Our task is to control LEDs. Here it makes sense to create a driver whose purpose will be to control LEDs. The driver will support up to 4 LEDs and will be able to control them. We need to be able to light, extinguish and change the state of the LED. Another function is to bind to a multiplexer (we will use it to indicate radio reception/transmission).

The driver files will be located in the drivers folder.

The device initialization code will initialize the driver, since we will need this driver in all cases to give an indication that something is happening. Therefore, it makes sense to add it to the initialization code, and not initialize it in every application that we will write. To control the LEDs we will use the PIN library module. To use this module we must connect the necessary headers and functions, add them to the ti-lib.h file.

```
 /*---------------------------------------------------------------------------*/
 /* ioc.h */
 #include DeviceFamily_constructPath(driverlib/ioc.h)

/*---------------------------------------------------------------------------*/
/* pwr_ctrl.h */
#include DeviceFamily_constructPath(driverlib/pwr_ctrl.h)

/*---------------------------------------------------------------------------*/
/* drivers/PIN.h */
#include <ti/drivers/PIN.h>
#include <ti/drivers/pin/PINCC26XX.h>

/*---------------------------------------------------------------------------*/
/* drivers/Power.h */
#include <ti/drivers/Power.h>
#include <ti/drivers/power/PowerCC26XX.h>
```

/*---*/

```
/* drivers/PIN.h */
#define ti_lib_driver_pin_init(...)                      PIN_init(__VA_ARGS__)
#define ti_lib_driver_pin_open(...)                      PIN_open(__VA_ARGS__)
#define ti_lib_driver_pin_close(...)                     PIN_close(__VA_ARGS__)
#define ti_lib_driver_pin_set_output_value(...)          PIN_setOutputValue(__VA_ARGS__)
#define ti_lib_driver_pin_get_output_value(...)          PIN_getOutputValue(__VA_ARGS__)
#define ti_lib_driver_pin_set_output_enable(...)         PIN_setOutputEnable(__VA_ARGS__)
#define ti_lib_driver_pin_get_input_value(...)           PIN_getInputValue(__VA_ARGS__)
#define ti_lib_driver_pin_get_config(...)                PIN_getConfig(__VA_ARGS__)
#define ti_lib_driver_pin_set_config(...)                PIN_setConfig(__VA_ARGS__)
#define ti_lib_driver_pin_set_mux(...)                   PINCC26XX_setMux(__VA_ARGS__)
#define ti_lib_driver_pin_set_wakeup(...)                PINCC26XX_setWakeup(__VA_ARGS__)
#define ti_lib_driver_pin_register_int_cb(...)           PIN_registerIntCb(__VA_ARGS__)

/*---------------------------------------------------------------------------*/
/* drivers/Power.h */

#define ti_lib_driver_power_init(...)                    Power_init(__VA_ARGS__)
#define ti_lib_driver_power_idle(...)                    Power_idleFunc(__VA_ARGS__)
#define
ti_lib_driver_power_inject_calibration(...)          PowerCC26XX_injectCalibration(__VA_ARGS_
_)
#define ti_lib_driver_power_shutdown(...)                Power_shutdown(__VA_ARGS__)
```

Also, to use this module you need to create constants (PINCC26XX Hardware attributes and PIN_Config). Since each device may have different I/Os that need to be initialized at startup, the PIN_Config object will be located in the device file. In our case this is BoardGpioInitTable which is empty.

```
/*
```

```
 *  ================================== PIN ==================================
 */
extern const PIN_Config BoardGpioInitTable[];
const PINCC26XX_HWAttrs PINCC26XX_hwAttrs = { .intPriority = ~0, .swiPriority = 0 };

/*
 *  ================================= Power =================================
 */
const PowerCC26XX_Config PowerCC26XX_config = {
    .policyInitFxn = NULL,
    .policyFxn = &PowerCC26XX_standbyPolicy,
    .calibrateFxn = &PowerCC26XX_calibrate,
    .enablePolicy = true,
    .calibrateRCOSC_LF = true,
    .calibrateRCOSC_HF = true
};
```

```
/*
 *  =============================== PIN ===============================
 */
const PIN_Config BoardGpioInitTable[] = {
     PIN_TERMINATE
};
```

Our application will light and extinguish LEDs at different frequencies. For this we also need the delay function, which is located in the NoRTOS module. This feature puts the device into sleep mode while waiting, so we will also need a power saving control module. For it, just like for the PIN module, you should connect the headers and add the necessary constants (PowerCC26XX_config).

Now the device initialization will look like this: Power Saving Initialization, PIN Module (GPIO) Initialization, and LEDs Driver Initialization.

```
void board_initialize(void)
{
    /* Initialize power */
    ti_lib_driver_power_init();

    /* Initialize pins */
    if (ti_lib_driver_pin_init(BoardGpioInitTable) != PIN_SUCCESS) {
        /* Error with PIN_init */
        while (1);
    }

    /* Initialize LEDs */
    leds_init();
}
```

In the LEDs driver, we create an array of LEDs (driver_leds_leds), as well as an array of pin configurations (driver_leds_pin_table). We configure each pin as an output in the PushPull configuration (it's clear why, we want to get the value 0 or 1) with an initial disabled state (PIN_GPIO_LOW). Next, the initialization function initializes the pins and puts them in the initial state.

LED control functions change the state of the selected pin to the desired position. It is worth paying attention to the toggle function (state change), we read the output state and write the opposite.

We will also need delays between changes in the status of the LEDs. They can be used in various ways. Due to the fact that microcontrollers are often powered by regular batteries, we should save energy and during periods when the microcontroller is not busy or a certain microcontroller module is not in use, we should turn it off, in other words, put it in sleep mode. Delay is a perfect example of such a situation. Let's add the following macros to the board_common.h file:

```
#define SLEEP_SECONDS(x) { rfn_posix_sleep(x); }
#define SLEEP_USECONDS(x) { rfn_posix_usleep(x); }
#define SLEEP_MSECONDS(x) SLEEP_USECONDS(x * 1000)
#define SLEEP_BREAK() { rfn_posix_sleep_break(); }
```

These macros put the microcontroller into sleep mode for a specified period of time. We will use them in the intervals between changing the status of the LEDs.

Let's build the application and launch it. We see that the LEDs light up and go out with a frequency of 100 milliseconds to 5 seconds in increments of 100 milliseconds.

Watchdog

Embedded systems often experience failures that can cause the system to freeze. For such cases, a watchdog mechanism has been created. Its meaning is that the program informs the guard that it is functioning. This is done by changing the logic value of its pin in the case of a hardware solution, or by calling the reset function at the software level. If the change has not occurred within a specified period of time, the guard restarts the system.

Let's create a watchdog driver. We will use the Watchdog library module.

Add the necessary headers to the ti-lib.h file.

```
/*---------------------------------------------------------------------------*/
/* watchdog.h */
#include DeviceFamily_constructPath(driverlib/watchdog.h)
/*---------------------------------------------------------------------------*/
/* drivers/Watchdog.h */
#include <ti/drivers/Watchdog.h>
#include <ti/drivers/watchdog/WatchdogCC26XX.h>
```

Let's create the necessary constants in the driver:

```
/*
 *  =============================== Watchdog ================================
 */
WatchdogCC26XX_Object watchdogCC26XXObjects[WATCHDOGCOUNT];

const WatchdogCC26XX_HWAttrs watchdogCC26XXHWAttrs[WATCHDOGCOUNT] =
{
    {
        .baseAddr = WDT_BASE,
        .reloadValue = 1000 /* Reload value in milliseconds */
    },
};

const Watchdog_Config Watchdog_config[WATCHDOGCOUNT] =
{
    {
        .fxnTablePtr = &WatchdogCC26XX_fxnTable, .object = &watchdogCC26XXObjects[WATCHDOG0],
        .hwAttrs = &watchdogCC26XXHWAttrs[WATCHDOG0]
    },
};

const uint_least8_t Watchdog_count = WATCHDOGCOUNT;
```

As you can see, we are setting the time period to 1 second. This means that if the watchdog has not been notified within a second, he will restart the system.

We will also add guard initialization to the device initialization code.

```
void board_initialize(void)
{
    /* Initialize power */
    Power_init();

    /* Initialize watch-dog */
    watchdog_initialize();

    /* Initialize pins */
    if (PIN_init(BoardGpioInitTable) != PIN_SUCCESS) {
        /* Error with PIN_init */
        while (1);
    }

    /* Initialize LEDs */
    leds_init();
}
```

It is worth noting that when the device goes into sleep mode, the watchdog continues to work. Therefore, you should pause system restarts during sleep. For this we will use the flag (watchdog_disabled). Let's add disabling the watchdog to the sleep mode macro.

```
#define SLEEP_SECONDS(x) { watchdog_enable(false); sleep(x); WDT_RESET(); watchdog_enable(true);
}
#define SLEEP_USECONDS(x) { watchdog_enable(false); usleep(x); WDT_RESET();
watchdog_enable(true); }
```

Also, in debugging mode, if we stop the program, the guard will continue to work, which will lead to a break in the debugging process. In this case, a watchdog mode is provided, which also stops it. To enable or disable it, we will use the WATCHDOG_STALL preprocessor directive during watchdog initialization.

```
#ifndef WATCHDOG_STALL
   watchdogParams.debugStallMode = Watchdog_DEBUG_STALL_OFF;
#endif
```

Also, for convenience, we will create a file (common_utilities) of auxiliary functions, such as time since system startup (get_up_time, get_up_time_us), or soft restart (reboot).

Add the necessary headers to the ti-lib.h file.

```
/*---------------------------------------------------------------------------*/
/* aon_rtc.h */
#include DeviceFamily_constructPath(driverlib/aon_rtc.h)

/*---------------------------------------------------------------------------*/
/* sys_ctrl.h */
#include DeviceFamily_constructPath(driverlib/sys_ctrl.h)

/*---------------------------------------------------------------------------*/
/* drivers/dpl/HwiP.h */
#include <ti/drivers/dpl/HwiP.h>
```

Please note the following code:

```
DEFINE_CRITICAL();
ENTER_CRITICAL();
…
EXIT_CRITICAL();
```

This code guarantees us atomic execution of the code contained inside. Atomicity is achieved by canceling interrupts. Therefore, it is important to perform a minimum of operations inside an atomic block so as not to miss an interrupt.

To obtain the time since system startup, we use the RTC module (aon_rtc in the library). Essentially, this is a timer that starts when the system starts.

To test the functionality of the watchdog, we will create a test application that is based on the previous test with the only difference being that after resetting the hibernation time, the application will stop notifying the watchdog and thereby lead to a system restart. It is important to understand that the guard works even while the microcontroller is in sleep mode. Therefore, we must disable system restarts during sleep. To do this, let's change the sleep macros.

```
#define SLEEP_SECONDS_WITH_WDT_ON(x) { rfn_posix_sleep(x); }
#define SLEEP_USECONDS_WITH_WDT_ON(x) { rfn_posix_usleep(x); }
#define SLEEP_MSECONDS_WITH_WDT_ON(x) SLEEP_USECONDS(x * 1000)
#define SLEEP_SECONDS(x) { watchdog_enable(false); rfn_posix_sleep(x); WDT_RESET();
watchdog_enable(true); }
#define SLEEP_USECONDS(x) { watchdog_enable(false); rfn_posix_usleep(x); WDT_RESET();
watchdog_enable(true); }
#define SLEEP_MSECONDS(x) SLEEP_USECONDS(x * 1000)
#define SLEEP_BREAK() { rfn_posix_sleep_break(); }
```

As you can see in the watchdog_test.c file, we have created a stop_watchdog variable that will act as a flag. While it is not raised, we reset the guard counter by calling the WDT_RESET() function. When it is raised, we stop resetting the watchdog counter to zero.

Important point! When a debugger is connected, or when a watchdog is called while the device is in sleep mode, the device will freeze. To disable the debugger, remove jumpers TMS, TCK, TDI, TDO, SWO.

Serial input output - universal asynchronous transceiver (UART)

One of the ways to exchange information is UART. This is a serial protocol that has a couple of lines of information for its transmission. Due to the fact that there are two lines, information can be transmitted in both directions simultaneously (asynchronously). Information is transmitted bit by bit over equal periods of time, which is determined by the transmission rate. The receiver is designated RX and the transmitter TX. Accordingly, two devices are connected to each other RX to TX.
We will use UART as the application console/terminal. To do this, let's create a Console driver.

Add the necessary headers to the ti-lib.h file.

```
/*---------------------------------------------------------------------------*/
/* drivers/utils/RingBuf.h */
#include <ti/drivers/utils/RingBuf.h>

/*---------------------------------------------------------------------------*/
/* drivers/UART.h */
#include <ti/drivers/UART.h>
#include <ti/drivers/uart/UARTCC26XX.h>
```

We will also create the necessary constants:

```
/**
 *  @def     UARTName
 *  @brief   Enum of UARTs
 */
typedef enum  UARTName {
     UART0 = 0,
     UARTCOUNT
}  UARTName;
```

```
/*
 *  ================================ UART =================================
 */

static UARTCC26XX_Object uartCC26XXObjects[UARTCOUNT];

static uint8_t uartCC26XXRingBuffer[UARTCOUNT][64];

const UARTCC26XX_HWAttrsV2 uartCC26XXHWAttrs[UARTCOUNT] = { {
        .baseAddr = UART0_BASE, .powerMngrId = PowerCC26XX_PERIPH_UART0, .intNum =
INT_UART0_COMB,
        .intPriority = ~0, .swiPriority = 0, .txPin = UART_TX, .rxPin =
        UART_RX,
        .ctsPin = PIN_UNASSIGNED, .rtsPin = PIN_UNASSIGNED, .ringBufPtr =
                uartCC26XXRingBuffer[UART0],
        .ringBufSize = sizeof(uartCC26XXRingBuffer[UART0]), .txIntFifoThr =
                UARTCC26XX_FIFO_THRESHOLD_1_8,
        .rxIntFifoThr = UARTCC26XX_FIFO_THRESHOLD_4_8 } };

const UART_Config UART_config[UARTCOUNT] = { { .fxnTablePtr = &UARTCC26XX_fxnTable, .object =
                                               &uartCC26XXObjects[UART0],
                                       .hwAttrs = &uartCC26XXHWAttrs[UART0] }, };

const uint_least8_t UART_count = UARTCOUNT;
```

Let's add a contact declaration for the UART interface to the device file.

```
/*---------------------------------------------------------------------------*/
/**
```

```
 * @brief   UART IOID mappings
 *
 * Those values are not meant to be modified by the user
 * @{
 */
#define  UART_TX                         IOID_3                  /* TXD */
#define  UART_RX                         IOID_2                  /* RXD */
#define  UART_CTS                        PIN_UNASSIGNED          /* CTS */
#define  UART_RTS                        PIN_UNASSIGNED          /* RTS */
/** @} */
```

The driver has two buffers, one for receiving and one for transmitting. They will be used for temporary storage of information. The main loop should call the console_process function to process the buffers and start transfers when necessary.

When reading data, we use a buffer because information can come in parts. And we need to somehow distinguish between the beginning and the end of the message. There are several ways to do this. One of them is to separate messages by a period of time. For example, messages cannot arrive with a period of less than 100 milliseconds. Another way to indicate the beginning and/or end of a message is with a special character. For example, the message ends with the symbol “CR” (Carriage Return). To do this, we will use the following preprocessor directives:

CONSOLE_RX_TIMEOUT_MS – Timeout in milliseconds between two subsequent messages.

CONSOLE_END_CHAR – End of message character.

If CONSOLE_END_CHAR is not defined, then CONSOLE_RX_TIMEOUT_MS will be used. In order to change CONSOLE_RX_TIMEOUT_MS from a batch file, you must set the preprocessor directive CONSOLE_CONF_RX_TIMEOUT_MS.

When the driver write function is called, the information is copied to a buffer and the transfer starts. If during recording the transfer was in progress, then at the end of the transfer, the driver will check whether there is data in the send buffer and if it is not empty, it will start a new transfer. According to the manufacturer's documentation, the UART_read and UART_write functions cannot be called from their callback functions, so they are called from the console_process function.

When incoming information is received, the read callback function is called, which is registered using the console_register_message_received_callback function.

To test the driver, let's create a console_test test that will perform the echo function. This means that the device will return the same information that it receives. It will also use a green LED to indicate reception/transmission (using the preprocessor directive CONSOLE_LED=LEDS_GREEN). This example uses "CR" to separate messages, so send a message ending with this character. Open a terminal (Docklight, Putty or any other application) and connect to the serial port named "XDS110 Class Application/User UART" with the following parameters:

BaudRate: 19200

Parity: None

DataBits: 8

StopBits: 1

Working with FLASH memory

Very often we need to change and save settings in an application. To do this we need access to persistent memory. Microcontrollers use FLASH memory for this purpose. It is important to note that the program itself is stored in the same memory, so it is important to use that part of the memory where the application is not located. Usually, for such purposes, a finite part of memory is used to avoid the program overlapping the settings.

The structure of FLASH memory is such that it is divided into pages of the same size. In CC1310 microcontrollers, the page size is 4kB. If the microcontroller memory is 120kB, then we have 32 pages of memory. The initial state of the memory bits is 1, and we can change it to 0 or leave it at 1. But if we changed a memory bit to 0 and want to change it back to 1, then this cannot be done. The only way to return a memory bit from 0 to 1 is to clear the page. In this case, we will lose all other state of the memory bits on this page. Also, FLASH memory has a limited number of erases. In other words, after a certain number of page clears, the memory may not work correctly. Typically, this number is in the tens of thousands. For greater vitality of FLASH memory, we should use memory page clearing, and use various algorithms to save the data we need. In CC1310 microcontrollers, the last 88 bytes of memory are used as so-called configuration registers, which set the operating parameters of the microcontroller. We will not touch on them in this course.

Let's look at the FLASH memory driver. It has the following features:

flash_load – This function reads a certain number of bytes from FLASH memory. Essentially, reading memory in a microcontroller is reading from a specific address. You can see that we are using the memcpy function for this purpose.

flash_erase_page – this function clears the memory page. It receives the page number as a parameter. Note that interrupts should be canceled during page erase, you can see that the page erase function call is placed in an atomic block:

```
ENTER_CRITICAL();
ret = ti_lib_flash_sector_erase(page_num * FLASH_PAGE_SIZE);
EXIT_CRITICAL();
```

Also, in CC1310 microcontrollers, part of the memory can be used for cache. In this case, you should disable the cache before clearing the page, and then turn it back on, which is what we do:

```
mode = ti_lib_vims_mode_get(VIMS_BASE);

if (mode != VIMS_MODE_DISABLED)
{
    /* Disable flash cache */
    ti_lib_vims_mode_set(VIMS_BASE, VIMS_MODE_DISABLED);
    while (ti_lib_vims_mode_get(VIMS_BASE) != VIMS_MODE_DISABLED);
}

.
.
.
```

```
.

if (mode != VIMS_MODE_DISABLED)
{
    /* Re-enable flash cache */
    ti_lib_vims_mode_set(VIMS_BASE, VIMS_MODE_ENABLED);
}
```

flash_write – this function writes a certain number of bytes into memory. Just like in the previous function, the write operation must be performed with interrupts and cache memory disabled. Please note that it does not clear the page, so before writing data, you should make sure that the page is cleared.

flash_load_ccfg – this function reads the configuration register data.

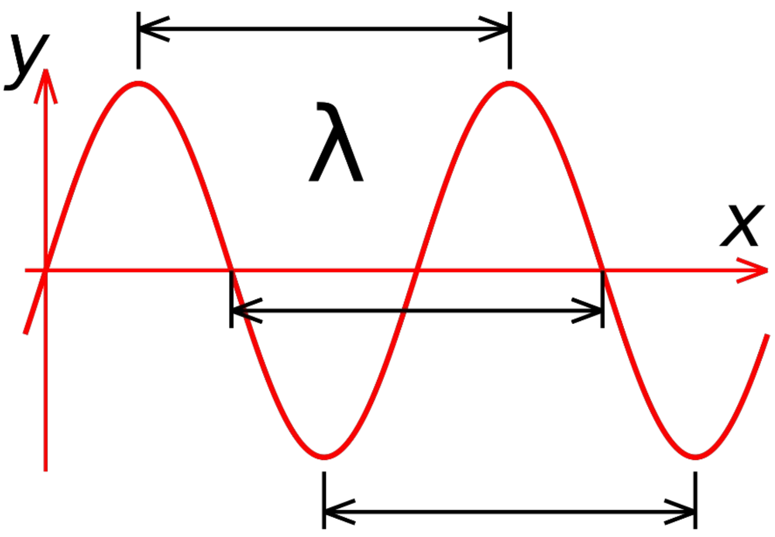
flash_save_ccfg – this function writes configuration registers.

To test the driver, we will create a test that will use page 30. The first step is to clear the page, then write the data, and the last step is to read the data and compare it with the original. We will output the results of each step to the console using the driver from the previous example. Since our test has several states, we will use a state machine. To do this, we use the step variable, which will show the current state. When the state is reached, the machine will remain in this state until restarted.

Also, in embedded systems it is customary to somehow notify the user about this at startup. The simplest solution is to use LED. When starting the program, we will blink all LEDs 2 times. To do this, we will use the **leds_blink_all** function of the LED driver.

Radio communication

In order to transmit data over the radio, you need to understand how this process generally works. First, let's remember what radio waves are and how we can use them to solve our problem. From the physics course we know that radio waves are electromagnetic vibrations. It is also known that electromagnetic waves propagate in a vacuum at the speed of light. Lightning is a natural source of radio waves. Artificially created radio waves are used for communication, and in our case, data transmission “over the air”. A receiver and transmitter are used to transmit and receive data. Often these two devices are in one, which can send and receive data. If we look at the wave we will see:



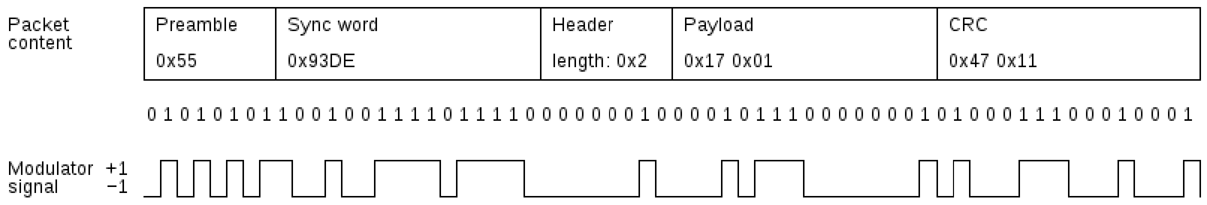
A wave has 3 main properties:

- **Амплитуда** – wave height
- **Частота** – wavelength (number of oscillations per unit time)
- **Фаза** – argument of a sine function, counted from the transition point of a negative value to a positive value and back

Using these properties, we will be able to transfer data. So how are these properties used? We all know that the “brains” of computers and other “smart” devices use binary logic. In other words, everything is built on ones and zeroes. All information is presented in the same format. Therefore, to transmit information, in essence, we need to transmit ones and zeros. Using one of the properties of radio waves, we can do this. In order not to go too far into theory, let's look at everything using an example. For example, let's take a property that we will use later, namely frequency, the same can be done with any other property. In other words, if we designate one frequency as 0 and the other as 1, then by changing the frequency we can transmit data over distances. This method is called “frequency modulation”, in English Frequency Modulation or FM. These two letters are probably familiar to you if you have ever used a radio. The simplest and most understandable “real life example” of frequency modulation is Morse code. It consists of short (dot) and long (dash) signals. In other words, signals of different frequencies.

Let's return to the world of IoT. We have two devices that are capable of receiving or sending data over a radio channel. To successfully transfer data, we need to configure them so that they use the same settings, otherwise our efforts will go to waste. One of the main parameters is the fundamental frequency. Let's say we use the frequency 433.92 MHz. This means that the frequencies we have chosen to represent zero and one will be on either side of the central one. Let's say 0 is a frequency of 433.90 MHz, and for 1 - 433.94 MHz. This deviation from the center frequency is called "Frequency Deviation". Since we are transmitting data bit by bit, in other words, sequentially, both sides must use the same transmit/receive data rate. Transmission speed is measured in baud (baud). And the last parameter is the transmit power, which is measured in dBm. To understand transmission power using an example, imagine that you are talking to a friend, and he gradually moves away from you. The further you move away, the louder you need to speak to hear each other.

The data itself is transmitted using packets up to 255 bytes in size. It is important to note that the longer the message, the longer it takes to transmit, which increases the chance that during transmission, another device will begin transmitting, causing interference, which could result in data loss. Therefore, it is advisable to transmit the smallest possible packets. Each package has the following structure:



- Preamble – the sequence 0101... or 1010... is used to indicate the start of transmission
- Sync word – a sequence of bytes (up to 4 bytes) to indicate the beginning of a message. The receiver uses the preamble to recognize the beginning of the transmission and does not always fully receive it; for this purpose, Sync word exists to know for sure where the beginning of the message is. This field can also be used to separate different systems by giving each a different Sync word.
- Header – basically contains the message size
- Payload – data we transmit
- CRC – checksum to verify received data and detect errors

It is also important to note that different countries have restrictions on the use of certain frequencies. Therefore, it is important to check what frequency you will use so as not to break the law. The Ministry of Communications is responsible for frequency distribution. In most countries, frequencies in the 433 MHz band are available for public use.

I think that the theory is enough to understand the basic principles of data transmission via radio. More detailed information can be easily found on the Internet. Let's get down to business, and in other words, let's look at the radio driver.

If we look at the block diagram of our microcontroller, we will see that a separate controller (Cortex M0) is responsible for the radio. It is connected to the main controller through a mechanism called a "doorbell". Its meaning is that the main controller sends commands to the radio controller, and when the operation is completed, it notifies the central controller through an interruption, as if ringing a doorbell. The radio controller supports a certain set of commands, using which we can reproduce reception, transmission and other operations. It also has a very accurate 4-megahertz clock. This watch can be used for a variety of tasks that require increased time accuracy.

First, let's look at the smartrf_settings. This file contains descriptions of the commands that the radio controller will use. Let's look at each command:

RF_cmdPropRadioDivSetup – this command is used to configure the radio. It contains information such as fundamental frequency, deviation, power, data transfer rate.

RF_cmdFs - this command is used to configure the radio signal synthesizer. It is used to generate a signal, so you need to set its operating frequency.

cmdRx – message receiving command. With its help we set the parameters and structure of the packet during reception.

cmdTx – message transmission command. With its help we set the parameters and structure of the packet during transmission.

RF_cmdRxTest – command to check the signal strength level during reception. With its help we can check the state of the air and determine the noise level.

RF_cmdTxTest – command to generate a signal. With its help, we can generate a constant “tone” signal of a certain frequency to calibrate the transmitter frequency.

cmdPropCs – transmission recognition command. With its help, we can determine the beginning of the transmission. For example, we need to send a data packet, but we don't want to interfere with other devices. Therefore, before starting to send a message, you can check whether there is a device that is currently transmitting a message, and if the airwaves are empty, you can send a packet. This technique is called "Listen before talk" or LBT.

cmdCountBranch – branch counter command. It is used in conjunction with the previous command to solve LBT. It's essentially a counter that decreases with each iteration until it reaches 0.

cmdNop – a command that does nothing. It is used for standby mode.

RF_cmdPropRxAdvSniff – sniffing command. Its meaning is to first recognize the "sniff out" of the preamble, and if it is recognized, continue receiving, otherwise stop. This method saves energy consumption.

By combining these commands, we can solve any problem.

It is also important to note that there is a product LAUNCHXL-CC13-90, which differs in that it has a signal amplifier. The amplifier itself is controlled using GPIO. Our driver will also support these devices.

So, let's look at setting up the radio module. It supports the following frequency bands:

- 433.05 - 434.79 MHz
- 865.0 - 867.0 MHz
- 863.0 - 870.0 MHz
- 868.0 - 868.6 MHz
- 868.7 - 869.2 MHz
- 869.4 - 869.65 MHz
- 869.7 - 870.0 MHz
- 902.0 - 912.0 MHz

And also the following data transfer rates:

- 4800
- 9600
- 19200
- 38400
- 57600
- 115200

To initialize, call the **radio_init** function. It receives the following parameters: data rate, frequency range, whether to use a signal booster or not, frequency deviation and bandwidth. The last 2 parameters can be set deviation=0xFFFF and rx_bandwidth=0xFF. In this case, they will be calculated automatically according to Carlson's capacity rule. This rule defines the relationship between deviation and throughput.

```
    /* Power amplifier is controlled by 3 pins, so we need to initialize them in case we are
using power amplifier */
    static PIN_Config ampPinTable[] = {
#if defined(Board_PIN_HGM) && Board_PIN_HGM != IOID_UNUSED
         Board_PIN_HGM | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
#endif
#if defined(Board_PIN_LNA_EN) && Board_PIN_LNA_EN != IOID_UNUSED
         Board_PIN_LNA_EN | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
#endif
#if defined(Board_PIN_PA_EN) && Board_PIN_PA_EN != IOID_UNUSED
         Board_PIN_PA_EN | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
#endif
         PIN_TERMINATE
    };
.
.
.


if (ampPinTable[0] != PIN_TERMINATE)
{
        /* In case we are using power amplifier (PA), try to initialize PA control pins */
        ampPinHandle = ti_lib_driver_pin_open(&ampPinState, ampPinTable);

        if (ampPinHandle == NULL) {
            /* Initialization failed */
            while(1);
        }
    }

    /* In case of using PA, set control pins initial state and route LNA, PA control pins to the
RF module */
#if defined(Board_PIN_HGM) && Board_PIN_HGM != IOID_UNUSED
    ti_lib_driver_pin_set_output_value(ampPinHandle, Board_PIN_HGM, (enable_pa)? 1 : 0);
#endif

#if defined(Board_PIN_LNA_EN) && Board_PIN_LNA_EN != IOID_UNUSED
    ti_lib_driver_pin_set_mux(ampPinHandle, Board_PIN_LNA_EN, PINCC26XX_MUX_RFC_GPO0);
#endif

#if defined(Board_PIN_PA_EN) && Board_PIN_PA_EN != IOID_UNUSED
    ti_lib_driver_pin_set_mux(ampPinHandle, Board_PIN_PA_EN, PINCC26XX_MUX_RFC_GPO1);
#endif
```

At the beginning of the radio_init function, we check the parameters and, if an amplifier is used, initialize the necessary GPIOs. Next, if we use an LED to indicate the activity of the radio module, we bind it to the output of the radio module.

```
#ifdef RADIO_ACTIVITY_LED
    /* If radio led is defined, bind it to the radio module. Actually we are routing IO to the
RF module output pin */
    leds_single_set_mux(RADIO_ACTIVITY_LED, PINCC26XX_MUX_RFC_GPO2);
#endif
```

Next, we initialize the queue of incoming packets.

```
    /* Initialize radio data message RX queue. This is actually radio module RX buffer */
    if (radio_rf_queue_define_queue(&radio_driver_rf_data_queue,
                                    radio_driver_rx_data_entry_buffer,
                                    sizeof(radio_driver_rx_data_entry_buffer),
                                    RF_QUEUE_NUM_DATA_ENTRIES,
                                    (RADIO_MAX_PACKET_LENGTH + RF_QUEUE_NUM_APPENDED_BYTES +
1)))
    {
        /* Store error code and wait for watchdog reset */
        while(1);
    }
```

Initializing radio parameters.

```
/* Initialize radio parameters */
    ti_lib_driver_rf_params_init(&radio_driver_rf_params);
    cmdRx.pQueue = &radio_driver_rf_data_queue;
    cmdPropCs.pNextOp = (rfc_radioOp_t*) &cmdTx;

    radio_driver_rf_baudrate = baudrate;
    radio_driver_rf_band = freq;
    /* Set speed */
    RF_cmdPropRadioDivSetup.symbolRate.rateWord = ((uint64_t)radio_driver_rf_baudrate *
0xF00000ULL / 0x16E3600ULL) & 0x1FFFFF;

    /* Set frequency */
    if (radio_get_available_channels_number(freq, baudrate, &radio_driver_base_frequency,
&radio_driver_frequency_channel_step) == 0)
    {
        /* No available channels */
        while(1);
    }
    /* According to required frequency, we need to use one of two available frequency ranges
(Low or High) */
    RF_cmdPropRadioDivSetup.loDivider = (freq == BASE_FREQUENCY_433)? 0x0A : 0x05;
    RF_cmdPropRadioDivSetup.pRegOverride = (freq == BASE_FREQUENCY_433)? pOverrides :
pOverrides868;
    radio_driver_tx_powers = (freq == BASE_FREQUENCY_433)? radio_driver_tx_power_table :
((radio_enable_pa)? radio_driver_tx_power_table_779_930_pa :
radio_driver_tx_power_table_779_930);

    RF_cmdPropRadioDivSetup.centerFreq = (uint16_t)(radio_driver_base_frequency / 100);

    /*
     * Optimal Deviation & BW filter
     *
     * Carson's bandwidth rule
     * DataRate = 2 x Deviation
     * OBW ~ DataRate + 2 x Deviation
     */
    switch (baudrate)
    {
    case RADIO_BAUDRATE_4800:
    case RADIO_BAUDRATE_9600:
        /*
         * Deviation 5kHz, OBW 39kHz
         */
        RF_cmdPropRadioDivSetup.modulation.deviation = 0x14; /* 5 << 2 */
        RF_cmdPropRadioDivSetup.rxBw = 0x20;
        break;
    case RADIO_BAUDRATE_19200:
        /*
         * Deviation 10kHz, OBW 49kHz
         */
        RF_cmdPropRadioDivSetup.modulation.deviation = 0x28; /* 10 << 2 */
        RF_cmdPropRadioDivSetup.rxBw = 0x21;
        break;
    case RADIO_BAUDRATE_38400:
        /*
         * Deviation 20kHz, OBW 78kHz
         */
        RF_cmdPropRadioDivSetup.modulation.deviation = 0x50; /* 20 << 2 */
        RF_cmdPropRadioDivSetup.rxBw = 0x23;
        break;
    case RADIO_BAUDRATE_57600:
        /*
         * Deviation 30kHz, OBW 98kHz
         */
        RF_cmdPropRadioDivSetup.modulation.deviation = 0x50; /* 30 << 2 */
        RF_cmdPropRadioDivSetup.rxBw = 0x24;
        break;
    case RADIO_BAUDRATE_115200:
        /*
         * Deviation 60kHz, OBW 236kHz
         */
        RF_cmdPropRadioDivSetup.modulation.deviation = 0x50; /* 60 << 2 */
        RF_cmdPropRadioDivSetup.rxBw = 0x28;
        break;
    default:
        break;
    }

    if (deviation != 0xFFFF)
    {
        /* Apply custom deviation if provided */
        RF_cmdPropRadioDivSetup.modulation.deviation = deviation;
    }
    if (rx_bandwidth != 0xFF)
    {
        /* Apply custom RX bandwidth if provided */
        RF_cmdPropRadioDivSetup.rxBw = rx_bandwidth;
    }
```

To use the radio, use the radio_open function. It is used once at program startup. Essentially, it starts configuring the radio module according to the parameters we set in the radio_init function.

The **radio_standby** function puts the radio module into power saving mode. In this mode, receiving and transmitting messages is impossible.

The **radio_close** function closes the driver and frees resources. After calling it, we should call the radio_open function if we need to use the module again.

The functions **radio_register_message_received_callback, radio_register_message_sent_callback, radio_register_command_completed_callback, radio_register_rssi_measured_callback** are used to track driver events.

The **radio_set_channel** function is used to set a channel in a frequency range. Essentially, it sets the frequency in the selected range.

The **radio_set_tx_power** function sets the signal transmission power level. There are also functions **radio_decrease_tx_power** and **radio_increase_tx_power**, which change the power level by one position.

The **radio_set_sync_word** function defines the Sync word.

The **radio_get_time** function is used to obtain the time of the radio module.

Let's look at the functions of receiving and transmitting packets. Packet reception is implemented in the **radio_enable_receive_internal** function.

It has the following parameters:

start_time – reception start time (time according to the radio module clock).

timeout_us – timeout in microseconds.

receive_on_timeout – continue receiving if the timeout is reached and the packet is partially received.

use_start_time – use start time or start immediately.

continue_on_receive – continue receiving if the packet is received and the timeout is not reached.

To begin with, it stops any activity (reception or transmission) if it is not completed.

```
/* Disable all active RX or TX commands */
RADIO_STOP_ACTIVITY();
```

Next, set the reception parameters.

```
if (use_start_time)
{
    diff = RAT_TIME_DIFF(start_time, radio_get_time());
    if (diff < 1)
    {
        /* Start time is missed */
        use_start_time = false;
        if ((timeout_us) && (ti_lib_driver_rf_convert_rat_ticks_to_us(-diff) < timeout_us))
        {
            timeout_us -= ti_lib_driver_rf_convert_rat_ticks_to_us(-diff);
        }
    }
}
/* Subscribe reader */
cmdRx.pNextOp = (RF_Op*) &cmdRx;
cmdRx.condition.rule = (timeout_us)? ((!continue_on_receive)? COND_NEVER: COND_STOP_ON_FALSE) :
COND_ALWAYS;
cmdRx.startTrigger.triggerType = (use_start_time)? TRIG_ABSTIME : TRIG_NOW;
cmdRx.startTime = (use_start_time)? start_time : 0;
cmdRx.startTrigger.pastTrig = 1;
cmdRx.maxPktLen = RADIO_MAX_PACKET_LENGTH + 1;
cmdRx.pktConf.bUseCrc = 1;
cmdRx.endTime = ti_lib_driver_rf_convert_us_to_rat_ticks(timeout_us);
/* Receive till the end if has timeout */
cmdRx.pktConf.endType = (!timeout_us && !receive_on_timeout)? 1 : 0;
cmdRx.endTrigger.triggerType = (timeout_us)? TRIG_REL_START : TRIG_NEVER;
cmdRx.status = IDLE;
```

And the reception starts.

```
/* Start RX */
radio_driver_rx_cmd_handle = ti_lib_driver_rf_post_cmd(radio_driver_rf_handle, (RF_Op*) &cmdRx,
RF_PriorityNormal, &radio_receive_callback,
                                         RF_EventCmdDone | RF_EventRxOk | RF_EventRxEntryDone);
if (radio_driver_rx_cmd_handle == RF_ALLOC_ERROR)
{
    /* Receive failed */
    radio_driver_rx_cmd_handle = 0;
    return false;
}
```

Sending packets is implemented in the **radio_send_internal** function.

It has the following parameters:

data – Pointer to the array of transmitted data.

size – Transmitted data size.

timestamp – Transmission start time (time according to the radio module clock).

lbt_rssi – Noise level for determining transmission in case of LBT.

lbt_timeout_us – LBT timeout in microseconds. Noise check time. 0 – do not use LBT.

use_start_time – use start time or start immediately.

To test the driver, we will create two tests. The first one will send data packets with a packet frequency of 5 seconds. When sending is complete, a message is displayed on the terminal with the number of messages sent. The second test will receive packets and report through the terminal the number of packets received. Also, when the radio is active, the red LED will light up.

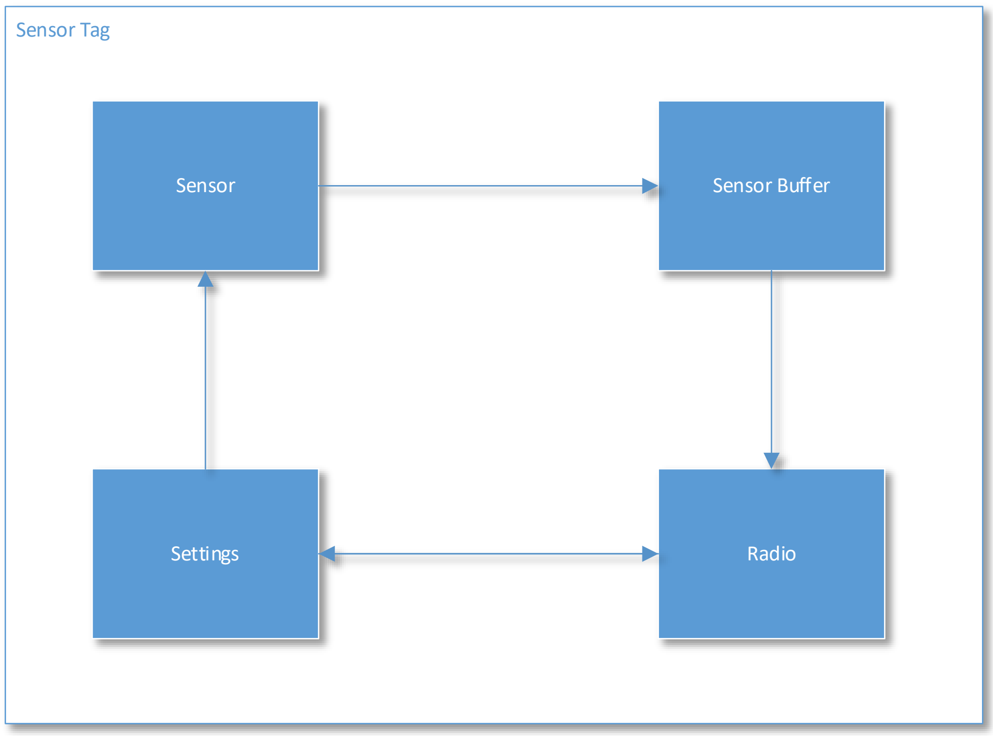
IoT project

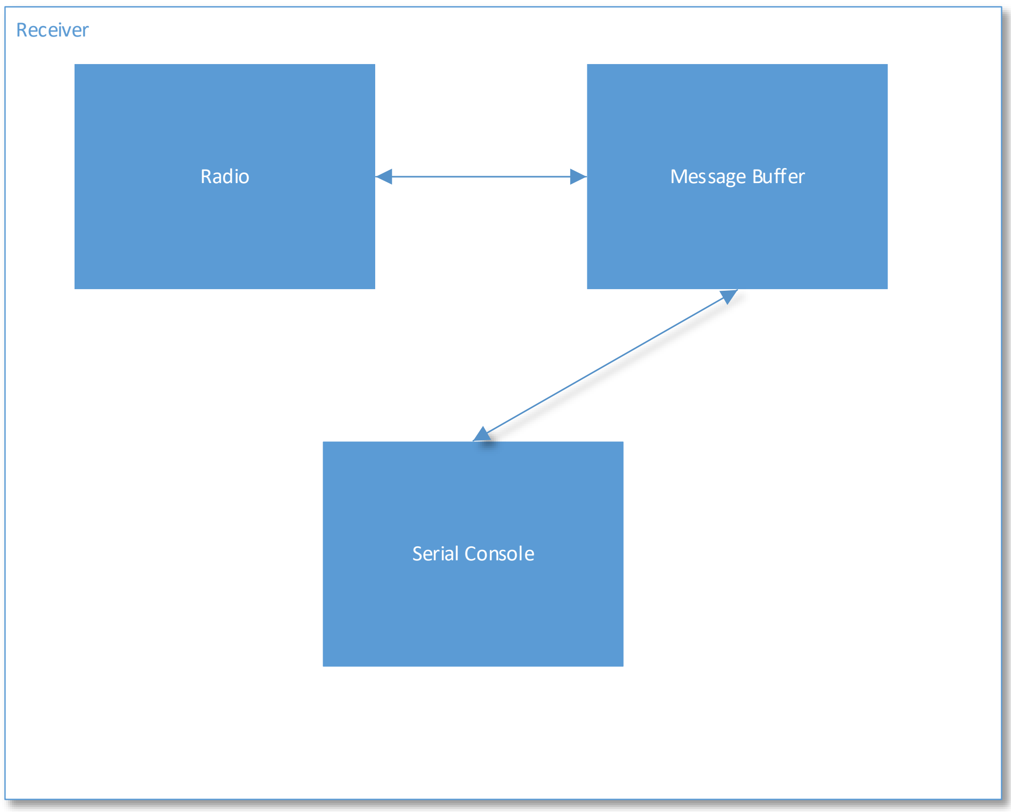
To get a feel for how IoT works, we will create a project. The idea behind the project is that we have wireless sensors/devices that are battery powered and send information at a certain period of time. There is also a base/central device that is connected to a computer, which receives data from sensors and outputs the received data through a terminal. It is also possible to change the reporting period of the sensor, in other words, how often the sensor will send data. For simplicity, the sensor will send the battery voltage (or supply voltage in cases where it is not powered by a battery). It will also be possible to send sensor data when you press a button on it.

Project design

Many young programmers can't wait to jump into programming. But don't rush. It is important to first think through the structure of the project, think about what modules it will consist of and how these modules interact with each other. Dividing a large project into small parts will give us the opportunity to better think through and implement each part. This will help you avoid those moments when you have to redo everything, and all the work done will go down the drain.

So, we have two types of devices, that is, two applications. The first is the receiving device, let's call it Gateway, and the second is the sensor itself. The Gateway must be constantly receiving so as not to miss data from various sensors. Therefore, it cannot enter energy saving mode and be powered by a battery. On the other hand, the sensor is in “sleep” mode most of the time and wakes up only when it needs to send data. For ease of understanding, we depict the modules and connections between them in the following block diagram.





All modules must work independently of others. This will give us the opportunity to check each module separately. The final application will link the modules. Each module must offer a convenient interface for working with it.

Application settings

Each application has settings that change as you use it. For example, the frequency with which the sensor measurements are taken, in our case this is the reporting period of the sensor. When the device is restarted, this parameter should not change, that is, it should be stored in permanent memory. The Settings module will be responsible for this functionality. The parameters will be stored as registers. This will give us the opportunity to simply add the necessary parameter in the future, if necessary.

Sensor buffer

The sensor buffer will store the data received from the sensor and, whenever possible, will be sent to the receiving device. The buffer has a queue structure, which means that the first element to enter the queue is also the first to leave it.

Message buffer

The Message buffer will store data received from sensors via radio and, whenever possible, will be sent to the console. The buffer has a queue structure, which means that the first element to enter the queue is also the first to leave it. This buffer also has 2 queues: received messages and messages to be sent.

Radio

This module (Radio) is responsible for transmitting data via radio. It contains a state machine according to which data will be transmitted/received. Since there are two types of devices in our project, and each works differently than the other, we will need to change the operation of the state machine depending on the type of device. Also, the “Sensor” device must support power saving mode during idle time.

Console

This module (Serial Console) is responsible for transmitting data via the serial port between the receiving device and the user's computer. Just like the radio module, it contains a state machine.

Sensor

This module (Sensor) is responsible for receiving data from the sensor. It contains a state machine, which will start the process of reading the sensor as needed.

Project implementation

So, let's start implementing the project. In the process, we will implement modules and test them using tests. As we implement all the modules, we will create final applications.

Settings module

The settings data is stored in ROM, in our case it is FLASH. Due to the fact that the program is also located in FLASH memory, we will store the settings in the final memory addresses. CC1310 LaunchPad™ uses a microprocessor with 128 kB FLASH memory, that's 32 pages. We will not use the last page, since it contains a CCFG, and when the page is erased, the CCFG will be reset. Of course, this can be prevented by adding code, but in this course, it will be easier for us to use page 31 instead.

So, to store the settings, we will create the following structure:

```
struct ATTRIBUTE_PACKED_ALIGNED(1) unit_settings_s
{
    /* Anchor */
    uint64_t anchor;
    /* Sensor readout interval */
    uint32_t sensor_readout_interval_sec;
    /* Settings checksum needed for settings validation */
    uint16_t checksum;
};
```

This structure in the header has a so-called anchor (or landmark), by which we can find it in FLASH memory. At the end of the structure there are two bytes of a checksum; we need it to verify the integrity of the settings. Pay attention to the ATTRIBUTE_PACKED_ALIGNED(1) macro. It's in the ti-lib.h file and essentially contains a compiler attribute that indicates that the structure uses one byte alignment rather than 4 (the default). This means that in the case of default alignment, the size of the structure will be a multiple of 4 even if its size is one byte. The default alignment of 4 bytes is done for a reason, the reason for this is that the microcontroller is 32-bit and it is easier and faster for it to perform operations with variables of 32 bits (4 bytes).

To calculate the checksum, we will use the CRC-16/XMODEM algorithm. Of course, you can use any other algorithm, but we will focus on it. To calculate the checksum, create the function calculate_crc16_xmodem in the common_utilities file.

The module has 3 functions:

settings_load – loading settings from FLASH memory into the settings variable.

settings_save – saving settings from the settings variable to FLASH memory.

settings_set_default – apply default settings. This function does not save settings to FLASH memory.

Let's look at the settings_load function. It uses the settings_find_pointer function, which looks for the anchor address on the settings page. If the address is not found (outside the page), the settings_load function will return false, otherwise it will load the settings from FLASH memory into the structure, calculate the checksum and compare it with the amount stored in the structure itself. The result of the comparison will be the result of the function.

The next function is settings_save. It also uses the settings_find_pointer function. If it returns an address within the page, then we reset all bits of the structure of the memory stored in FLASH, advance the pointer to the next position and write the settings structure from the variable there. Otherwise, we clear the page and write the structure to the top of the page. Please note that if the pointer that the settings_find_pointer function returns is equal to the last possible position of the settings structure, then in this case we are clearing the page, because if we advance the pointer to the next position, we will go off the page. Also, please note that before writing to FLASH memory, we recalculate the checksum so that it remains updated.

The settings_set_default function sets the default settings to the settings variable and recalculates the checksum.

To test the module, let's create a test - settings_test. It is important for us to check the behavior of the module if we reach the end of the settings page. To do this, we will change the settings, save, and read them back. Then we compare the saved settings with those that were saved and display the result of the comparison on the console. We will repeat this procedure 300 times, since the structure is 14 bytes in size, and in 300 iterations we will fill the entire page (total number of bytes = 4200). Please note that in 300 iterations we erased the page only once, thereby extending the life of FLASH memory.

Sensor buffer

We will store the data received from the "sensor" in RAM using a cyclic buffer. In other words, we will use a queue that stores data received from the sensor. Essentially, we will write sensor_buffer_record_s structures to the queue. This structure looks like this:

```
struct ATTRIBUTE_PACKED_ALIGNED(1) sensor_buffer_record_s
{
    /* Header */
    struct header
    {
        /* Sensor ID */
        uint8_t sensor_id;
        /* Record size */
        uint8_t record_size;
    } record_header;
    /* Records array */
    uint8_t record[SENSOR_BUFFER_RECORD_MAX_SIZE];
};
```

It has a header containing the sensor code and the recording size. After the header, the data itself is stored. The maximum data size is 12 bytes. The sensor buffer size will be 300 bytes. These parameters can be easily changed later.

For the buffer itself we will use an auxiliary structure:

```
struct ATTRIBUTE_PACKED_ALIGNED(1) sens_buffer_s
{
    /* Ring buffer object */
    RingBuf_Object  buf_obj;
    /* Ring buffer records count */
    uint16_t buf_count;
};
```

It consists directly of a circular buffer object and the number of records stored.

The module will also have the following functions:

sensor_buffer_initialize – Buffer initialization.

sensor_buffer_get_records_count – Getting the number of entries in the buffer.

sensor_buffer_pop_record – Retrieving an entry from a buffer.

sensor_buffer_push_record – Adding an entry to the buffer.

Let's look at each of them. When initializing the buffer, we initialize the circular buffer and reset the counters.

sensor_buffer_get_records_count simply returns the quantity from the counter.

sensor_buffer_push_record –

```
bool sensor_buffer_push_record(sensor_buffer_record_union_t record)
{
    size_t i;
    static sensor_buffer_record_union_t tmp;

    /* Free buffer if needed */
    WDT_RESET();
    while (sensor_buffer.buf_obj.count &&
        ((sensor_buffer.buf_obj.length - sensor_buffer.buf_obj.count) <
MIN(sizeof(record.array), SENSOR_RECORD_HEADER_SIZE + record.record.record_header.record_size)))
    {
        /* Need to remove */
        sensor_buffer_pop_record(&tmp);
    }

    /* Put to buffer */
    for (i = 0; i < MIN(sizeof(record.array), record.record.record_header.record_size +
SENSOR_RECORD_HEADER_SIZE); i++)
    {
        if (!sensor_buffer_put_byte(record.array[i]))
        {
            return false;
        }
    }
    sensor_buffer.buf_count++;
    return true;
}
```

first checks whether there is enough space to add a new entry. If there is not enough space, records are taken out until there is enough free space. Next, it copies the bytes of the structure from the buffer, increases the number of entries and exits.

sensor_buffer_pop_record –

```
bool sensor_buffer_pop_record(sensor_buffer_record_union_p record)
{
    size_t i;
    if (record == NULL || sensor_buffer.buf_count == 0)
    {
        return false;
    }
    memset(record->array, 0, SIZE_OF_ARRAY(record->array));
    for (i = 0; i < SENSOR_RECORD_HEADER_SIZE; i++)
    {
        sensor_buffer_get_byte(record->array + i);
    }
    for (i = 0; i < MIN(SENSOR_BUFFER_RECORD_MAX_SIZE, record-
>record.record_header.record_size); i++)
    {
        sensor_buffer_get_byte(record->array + SENSOR_RECORD_HEADER_SIZE + i);
    }
    sensor_buffer.buf_count--;
    return true;
}
```

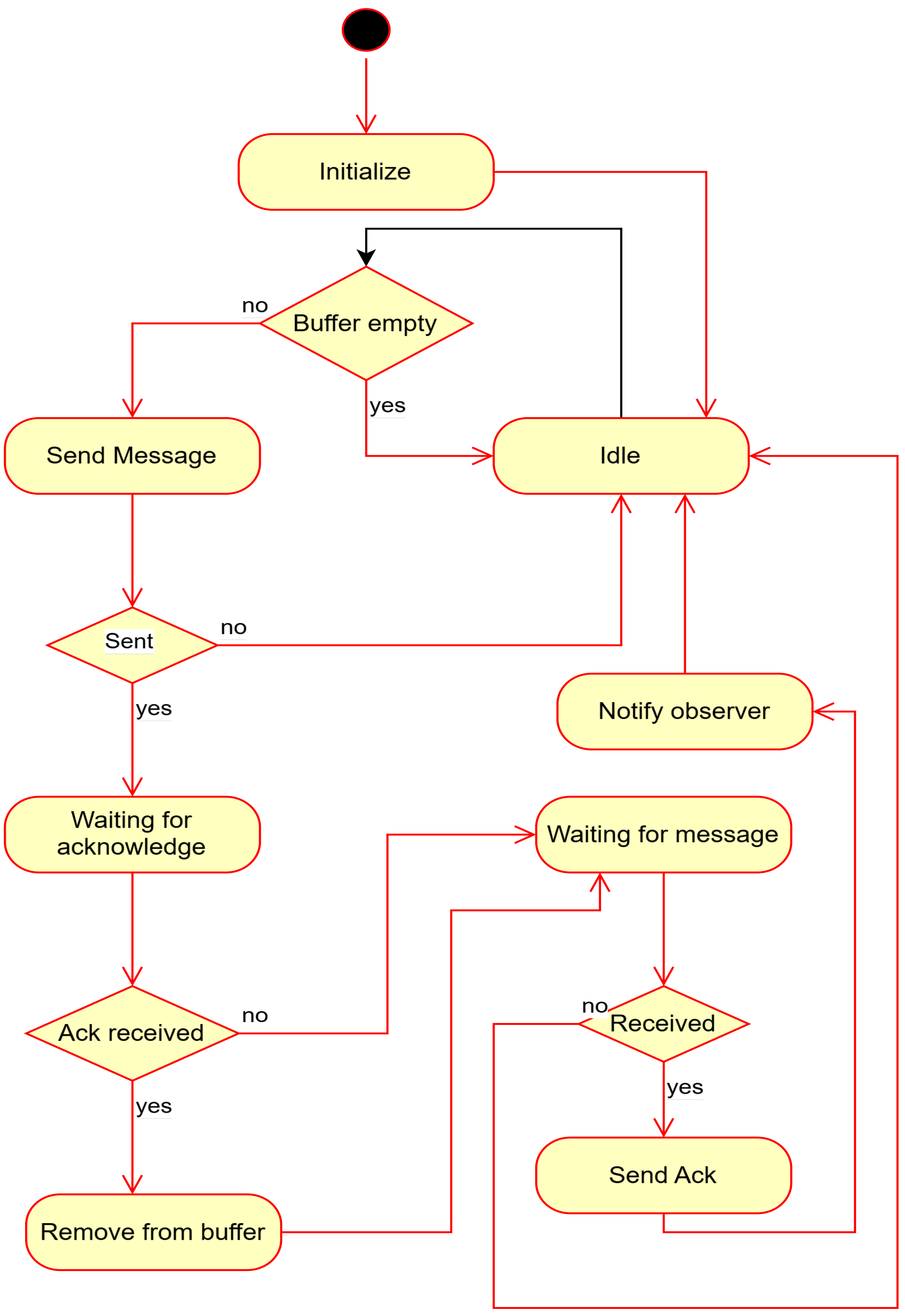
First, it checks whether there are records in the buffer; if there is, it first copies the header, and then the data, depending on the size of the data written in the header.

To test the module, let's create a sensor_buffer_test test. It will save data to a buffer and read it back. Then we compare the saved settings with those that were saved and display the result of the comparison on the console. We will repeat this procedure 300 times.

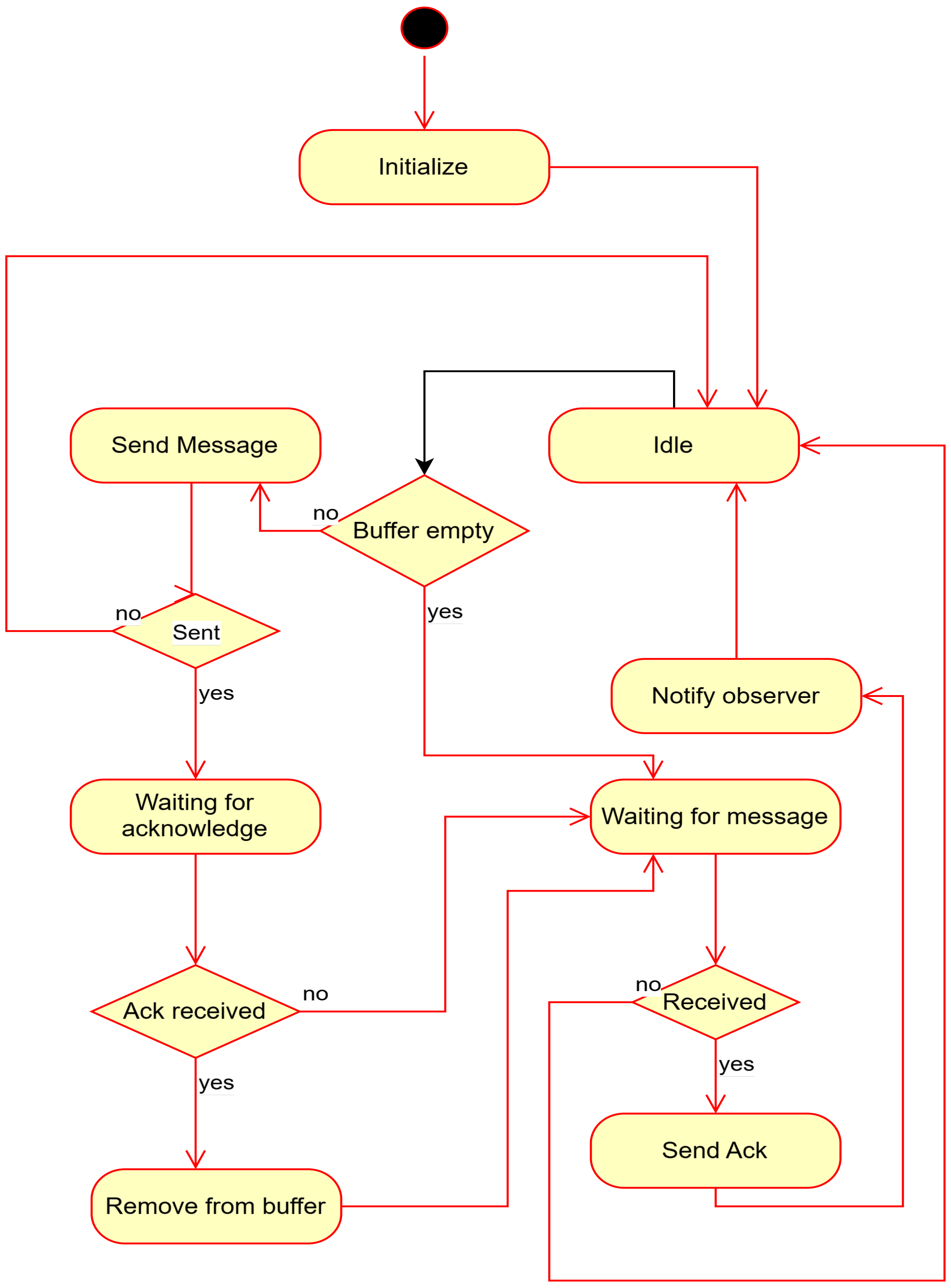
Radio Protocol

As we know from project design, we have two types of devices: a receiving device and a transmitter.

Let's consider the state machine of the transmitter. We have an outgoing message buffer into which the sensor will add messages. If it is empty, the machine will go into the waiting state (IDLE). In this state we can put the device into sleep mode. When a message appears in the buffer, the machine will switch to sending mode. If the sending is successful, the machine will go into the mode of receiving an acknowledgment message (Acknowledge) from the recipient; otherwise, it will go back to the standby mode. Upon receipt of Acknowledge, the machine will delete the message from the buffer. The machine will then go into message waiting mode. If a message is received, the machine will switch to Acknowledge sending mode, and then notify about the receipt of the message and return to standby mode.



The receiving device has a similar state machine, with the difference that it does not go into sleep mode, but instead goes into message waiting mode.



Now let's look at the message structure. Each message has a header, which is identical for all types of messages:

```
/**
 * @brief   Radio message structure
 */
typedef struct radio_message_header_s radio_msg_header_t, *radio_msg_header_p;

struct ATTRIBUTE_PACKED_ALIGNED(1) radio_message_header_s
{
    /* Message size from next byte to the end */
    uint8_t size;
    /* Message type */
    uint8_t msg_type;
    /* Source ID */
    uint32_t source_id;
    /* Destination ID */
    uint32_t dest_id;
    /* Message number counter */
    uint8_t msg_num;
};

/**
 * @brief   Radio message header union
 */
typedef union radio_msg_header_union_u radio_msg_header_union_t, *radio_msg_header_union_p;

union radio_msg_header_union_u
{
    radio_msg_header_t header;
    uint8_t array[11];
};
```

The header has the following fields:

- Message size in bytes
- Message type
- Sender device ID
- Receiver device ID
- Message number (required for Acknowledge)

To get the device ID we will use the MAC address. It consists of 8 bytes, but we will take the bottom 4, that will be enough. To get the MAC address, use the get_mac_address function.

Please note that we also created a union. This is a very useful "trick" when working with data arrays; to create a message we will use the structure field (header in the case of header), and when forwarding we will use the array field (array).

Also pay attention to the ATTRIBUTE_PACKED_ALIGNED(1) attribute. It is used to tell the compiler to align variables to a length that is a multiple of 1 byte, rather than the default 4 bytes. If we use 4 byte alignment, then the structures will occupy memory sizes that are a multiple of 4 bytes, and in the case of our header we have 11 bytes, which is not a multiple of 4.

The Acknowledge message looks as follows:

```
/**
 * @brief   Radio acknowledge message structure
 */
typedef struct radio_acknowledge_message_s radio_acknowledge_message_t,
*radio_acknowledge_message_p;

struct ATTRIBUTE_PACKED_ALIGNED(1) radio_acknowledge_message_s
{
    /* Header */
    radio_msg_header_t header;
    /* Acknowledge message number */
    uint8_t ack_msg_num;
};
```

```
/**
 * @brief   Radio acknowledge message header union
 */
typedef union radio_acknowledge_msg_union_u radio_acknowledge_msg_union_t,
*radio_acknowledge_msg_union_p;

union radio_acknowledge_msg_union_u
{
    radio_acknowledge_message_t acknowledge;
    uint8_t array[12];
};
```

As you can see, the message consists of a header and the number of the message it acknowledges.

In addition to the header, the Data message has an array of data. The maximum array size is set using RADIO_MAX_PAYLOAD_SIZE.

```
/**
 * @brief   Radio data message structure
 */
typedef struct radio_data_message_s radio_data_message_t, *radio_data_message_p;

struct ATTRIBUTE_PACKED_ALIGNED(1) radio_data_message_s
{
    /* Header */
    radio_msg_header_t header;
    /* Data */
    uint8_t data[RADIO_MAX_PAYLOAD_SIZE];
};

/**
 * @brief   Radio data message header union
 */
typedef union radio_data_msg_union_u radio_data_msg_union_t, *radio_data_msg_union_p;

union radio_data_msg_union_u
{
    radio_data_message_t data;
    uint8_t array[11 + RADIO_MAX_PAYLOAD_SIZE];
};
```

The command message looks as follows:

```
/**
 * @brief   Radio command message structure
 */
typedef struct radio_cmd_message_s radio_cmd_message_t, *radio_cmd_message_p;

struct ATTRIBUTE_PACKED_ALIGNED(1) radio_cmd_message_s
{
    /* Header */
    radio_msg_header_t header;
    /* Direction (Request - 0, Response - 1) */
    uint8_t direction;
    /* Command (Get - 1, Set - 2) */
    uint8_t command;
    /* Params */
    Uint8_t params[6];
};

/**
 * @brief   Radio command message header union
 */
typedef union radio_cmd_msg_union_u radio_cmd_msg_union_t, *radio_cmd_msg_union_p;
```

```
union radio_cmd_msg_union_u
{
    radio_cmd_message_t cmd;
    uint8_t array[19];
};
```

It has the following fields:

- Direction – Request or response to a request.
- command – Read or write.
- Параметры – command parameters. We will use a "lookup table" or in other words registers. Each register is responsible for some setting parameter. The parameters can be a register number and its value. Also, when responding to a request, we will return a result code (0 – error, 1 – OK).

To distinguish between two types of state machines in the code, we will use the ROOT_NODE macro.

Let's consider the main states of the state machine of the radio protocol. As you can see from the radio_protocol.c file, state switching occurs in the main function radio_protocol_process, or in the radio driver event handlers.

Initialization state – This state is started when the program starts. It tunes the radio module to the desired frequency and speed according to the macros specified at the beginning of the file or through the command file. We also connect the radio driver event handlers and put the machine in the standby state.

Waiting state - in this state we determine what the next command is. It depends on whether there are messages to be sent in the buffer (in this example, for simplicity, we will use a buffer of one message), if the buffer is not empty, then we transfer the machine to the sending state, otherwise to the receiving state (for the receiver) or in the standby state (for the sensor).

Reception state – in this state we start receiving messages. It is important to note that the transmitter has a timeout of one message. If no message is received during this time, the device will go into sleep mode.

Of course, this state machine is not ideal, since if Acknowledge is not received, it can send messages without stopping, but for educational purposes this is enough for us.

Sending state - in this state we start sending the message if there is one. Otherwise, the machine will go into the waiting state, and in the case of a receiver, into the receiving state. It is also worth noting that when sending a message we will use the LBT (Listen before talk) mechanism to avoid collisions between other devices, but sending Acknowledge without, because the response must be instantaneous.

Message processing state - in this state, the machine processes the received message and notifies the listener (main application) about receiving the message by calling the event handler.

We also have auxiliary functions:

- radio_protocol_is_active – shows whether the state machine is busy or you can go into sleep mode.
- radio_protocol_has_outgoing_message – is there a message to send?
- radio_protocol_append_data_message – adds a message of type Data to send.
- radio_protocol_append_command_message - adds a Command type message to send.
- As well as functions for registering event handlers.

Battery Monitor

From the project description, the device with the sensor must send battery voltage. To obtain the battery voltage, we will use the built-in microprocessor module (AON_BATMON), which allows us to obtain the input voltage of the microcontroller and its temperature. The module is implemented in the file bat_monitor_sensor. Due to the fact that voltage and temperature measurements do not occur instantly, the module will use a state machine so as not to interfere with the operation of other modules by blocking during measurement. The machine has only three states:

- BAT_MONITOR_SENSOR_STATE_SAMPLE – Sample start.
- BAT_MONITOR_SENSOR_STATE_SAMPLE_STARTED – Sample in process.
- BAT_MONITOR_SENSOR_STATE_NONE – Waiting.

When the measurement is completed, the module calls the event handler and passes the measurement result to it (measurement time from the start, voltage in millivolts and temperature in degrees).

To save energy, the module stops measuring AON_BATMON.

Push Button

This module is responsible for processing button clicks. In the project we will use the BTN-2 button of the development kit. The module is implemented in the file push_button. When the button is clicked and released, event handlers are called. It is important to note that button actuation works even in sleep mode, so if we want to send a message, we should wake the microcontroller from sleep mode using the SLEEP_BREAK macro.

Tag application (unit with sensor)

To create a device application with a sensor, we already have all the necessary modules. Let's look at the implementation of the application. Its code is in the tag.c file. Application initialization begins with loading settings.

```
    /* Load settings */
    if (!settings_load())
    {
        settings_set_default();
        settings_save();
    }
```

If the download fails, we apply the default settings.

Next, to notify the user about the launch of the program, we blink the LEDs twice. This practice is often used in embedded systems.

```
#if defined(RADIO_ACTIVITY_LED) || defined(CONSOLE_LED)
    /* Blink leds to indicate program start */
    leds_blink_all(2);
#endif
```

Next, we initialize the radio module, sensor buffer and sensor, and bind event handlers.

```
    /* Initialize Radio protocol */
    radio_protocol_process();
    radio_protocol_register_data_message_received_callback(&radio_protocol_data_message_received
_cb);
    radio_protocol_register_command_message_received_callback(&radio_protocol_command_message_re
ceived_cb);

    /* Initialize sensor */
    sensor_buffer_initialize();
    bat_monitor_sensor_init();
    bat_monitor_sensor_register_measurement_completed_callback(&bat_monitor_sensor_measurement_c
ompleted_cb);
    bat_monitor_sensor_process();
    push_button_init();
    push_button_register_reed_switch_released_callback(&push_button_released_cb);
```

In the main loop we call processing functions in the necessary modules and in cases of idle time we send the device to “sleep”. Sleep time depends on whether there are messages to send. If there are any, the time will be equal to 1 second.

```
    /* Main loop */
    while (1)
    {
        /* Reset watch-dog */
        WDT_RESET();

        /* Process radio protocol */
        radio_protocol_process();

        /* Process sensor */
        bat_monitor_sensor_process();

        /* Process messages */
        process_sensor_buffer();

        /* Sleep if needed */
        if (!radio_protocol_is_active() && !bat_monitor_sensor_get_prevent_low_power())
        {
            /* Can enter low power */
            SLEEP_MSECONDS(((radio_protocol_has_outgoing_message())? 1 :
settings.sensor_readout_interval_sec) * 1000);
        }
    }
```

When processing the sensor buffer, we check whether there are messages in it, and if it is not empty, we get the message and pass it to the radio module for sending.

```
    if (sensor_buffer_get_records_count())
    {
        /* Message exists, forward it to radio module */
        memset(&record, 0, sizeof(record));
        if (sensor_buffer_pop_record(&record))
        {
            /* Forward as data message */
            radio_protocol_append_data_message(0xFFFFFFFFU, record.record.record,
record.record.record_header.record_size);
        }
    }
```

We have a similar situation in the incoming command handler. We check if a request to read or write a register is received, then we execute the command, generate the result and transmit it to the radio module. It is worth noting that when writing one to register 0, the device will restart. This way we can remotely restart the device.

```
    /* Parse command and respond only to requests */
    if (cmd_msg->cmd.direction == 0)
    {
        if (cmd_msg->cmd.command == 1)
        {
            /* Get register */
            memcpy(&reg, cmd_msg->cmd.params, 2);
            memcpy(params, cmd_msg->cmd.params, 2);
            memset(params + 2, 0, 4);
            params[6] = 0;
            if (reg == 1)
            {
                memcpy(params + 2, &settings.sensor_readout_interval_sec, 4);
                params[6] = 1;
            }
            radio_protocol_append_command_message(0xFFFFFFFFU, 1, 1, params, 7);
        }
        else if (cmd_msg->cmd.command == 2)
        {
            /* Set register */
            memcpy(&reg, cmd_msg->cmd.params, 2);
            memcpy(&val, cmd_msg->cmd.params + 2, 2);
            memcpy(params, cmd_msg->cmd.params, 2);
            memset(params + 2, 0, 5);

            if (reg == 0 && val == 1)
            {
                /* Reboot */
                reboot();
                return;
            }
            else if (reg == 1)
            {
                settings.sensor_readout_interval_sec = val;
                params[2] = settings_save()? 1 : 0;
            }
            radio_protocol_append_command_message(0xFFFFFFFFU, 1, 2, params, 3);
        }
    }
```

When the button is clicked, the button's event handler will fire. In it we will launch a new voltage measurement and raise the flag. If the click occurred during “sleep”, we wake the microcontroller from sleep mode.

```
    /* Force measure voltage */
    bat_monitor_sensor_read(true);
    force_readout = true;
    SLEEP_BREAK();
```

And when the measurement is completed, we create an entry in the sensor buffer. If the flag is raised, we wake up the microcontroller from sleep mode and lower the flag.

```
    bat_monitor_sensor_result_p res = (bat_monitor_sensor_result_p)result;
    static sensor_buffer_record_union_t record;

    /* Put result to sensor buffer */
    memset(&record, 0, sizeof(record));
    record.record.record_header.sensor_id = 1; /* Battery monitor */
    record.record.record_header.record_size = 12;
    memcpy(record.record.record, &res->timestamp, 4);
    memcpy(record.record.record + 4, &res->voltage, 4);
    memcpy(record.record.record + 8, &res->temperature, 4);

    sensor_buffer_push_record(record);

    if (force_readout)
    {
        force_readout = false;
        SLEEP_BREAK();
    }
```

Console Protocol

The receiving device outputs the received data from the transmitters through the console. Commands received through the console are also used to configure transmitters. The console module is responsible for transmitting data through the console. As mentioned earlier, we will change the parameters of the sensors using register commands, namely reading and writing registers.

The read command has the following format:

GETREG:<Unit ID>,<Register><CR>

Example command:

GETREG:329563755,1<CR>

This command reads sensor register 1 with number 329563755. Register 1 is the sensor reading frequency in seconds.

The read command has the following format:

GETREG:<Unit ID>,<Register>,<Value><CR>

Example command:

SETREG:329563755,1,60<CR>

This command writes the value 60 to register 1 of sensor number 329563755.

It is also possible to restart the device remotely. To do this, write the value 1 to the device register 0.

Let's look at the implementation of the module. It is located in the console_protocol.c file. To initialize the module, use the console_protocol_initialize function. It configures the UART driver. Default Serial Port Settings:

Baudrate: 19200

Parity: None

Data Bits: 8

Stop Bits: 1

And also, during initialization we bind the event handler for received messages. It will parse the received message, and if it is a register read or write command, the event handler will be called to notify the main application.

There are also two functions for outputting results in text format.

console_protocol_send_command_message – displays the result of reading/writing registers.

console_protocol_send_sensor_data_message - displays the sensor reading result.

Receiver unit application (Gateway)

To create a device application that will receive sensor data, consider the gateway.c file. The initialization is similar to the initialization of the Tag application with the only difference that there is no initialization of the sensor, but instead initialization of the console module.

```
/* Load settings */
    if (!settings_load())
    {
        settings_set_default();
        settings_save();
    }

#if defined(RADIO_ACTIVITY_LED) || defined(CONSOLE_LED)
    /* Blink leds to indicate program start */
    leds_blink_all(2);
#endif

    /* Initialize Radio protocol */
    radio_protocol_process();
    radio_protocol_register_data_message_received_callback(&radio_protocol_data_message_received
_cb);
    radio_protocol_register_command_message_received_callback(&radio_protocol_command_message_re
ceived_cb);

    /* Initialize Console protocol */
    console_protocol_initialize();
    console_protocol_register_command_message_received_callback(&console_protocol_command_messag
e_received_cb);
```

In the main loop we call processing functions in the necessary modules. Please note that this application does not put the device into sleep mode.

```
    /* Main loop */
    while (1)
    {
        /* Reset watch-dog */
        WDT_RESET();

        /* Process radio protocol */
        radio_protocol_process();

        /* Process console */
        PROCESS_CONSOLE_PROTOCOL();
    }
```

Radio module event handlers call console module functions to notify the user of received messages. In the case when the message is the result of a command, it is checked if it is a response command (cmd_msg->cmd.direction == 1).

```
void radio_protocol_data_message_received_cb(radio_data_msg_union_p data_msg, int8_t rssi)
{
    console_protocol_send_sensor_data_message(data_msg->data.header.source_id, rssi, data_msg-
>data.data, data_msg->data.header.size - (sizeof(radio_data_message_t) - RADIO_MAX_PAYLOAD_SIZE
- 1));
}

void radio_protocol_command_message_received_cb(radio_cmd_msg_union_p cmd_msg, int8_t rssi)
{
    if (cmd_msg->cmd.direction == 1)
    {
        console_protocol_send_command_message(cmd_msg->cmd.header.source_id, rssi, cmd_msg-
>cmd.command, cmd_msg->cmd.params, cmd_msg->cmd.header.size - sizeof(radio_cmd_message_t) - 1);
    }
}
```

On the other hand, the console module event handler forwards the command to the radio module, which in turn will forward it to the desired device.

Conclusion

So, we created our first IoT project. Of course, it is far from ideal, but it is always worth starting with something simple and gradually complicating the tasks. I hope this project gives you a good start in the exciting world of IoT. I would be glad to receive your opinion about the work I have done, because it is always useful to receive feedback on the work done, even if this feedback is not entirely pleasant. You can contact me by mail: kulipator@gmail.com.