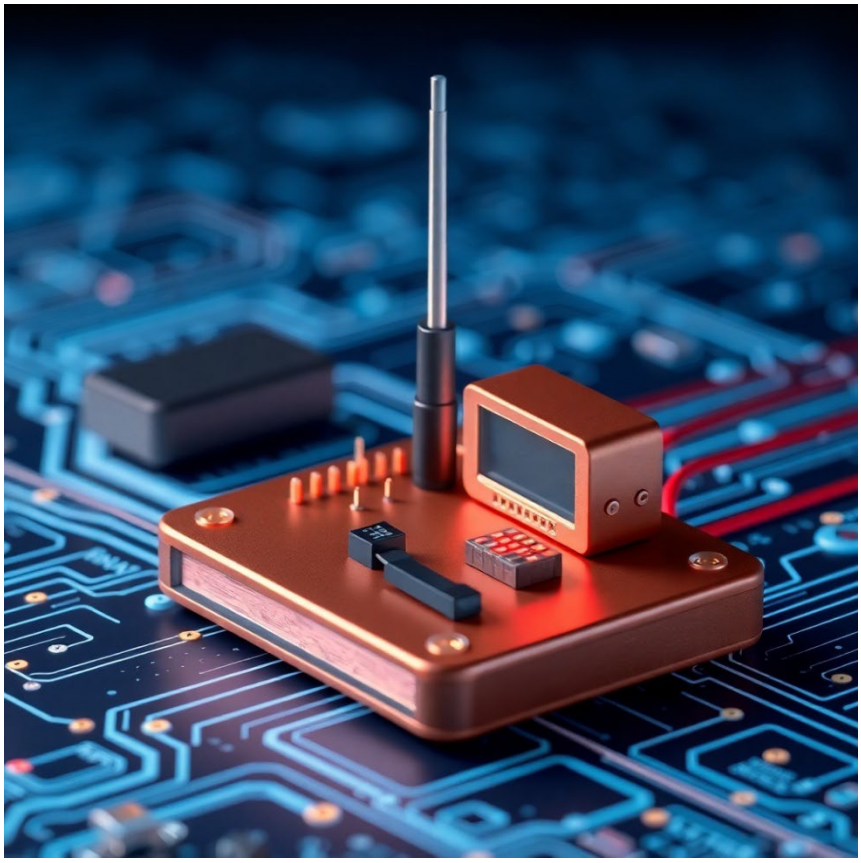


Программируем IoT своими руками

Содержание

Введение

Интернет вещей (IoT) был предсказан еще в далеком 1926 году Николой Теслой. В своем интервью одному из журналов ученый рассказал, что в будущем все физические предметы объединятся в огромную систему. В наши дни уже широко используются устройства, подключенные к мировой паутине. Одни собирают информацию о внешней среде, другие управляют процессами, их используют в различных отраслях. Вы наверняка сталкивались с примерами их использования в повседневной жизни начиная от «умного дома» и заканчивая умными системами полива полей или беспилотными разведывательными самолетами. Многие компании строят свой бизнес на создании умных устройств или представлении различных сервисов, связанными с интернетом вещей.

В этом учебнике я попробую поделиться с вами опытом разработки программного обеспечения таких устройств, а в частности беспроводных датчиков, целью которых является сбор информации с датчика и передачи данных по беспроводному каналу (радио). Поскольку одно из важных критериев этих устройств дешевизна, они используют микроконтроллеры для сбора и передачи информации. Курс научит вас программированию микроконтроллеров, в частности микроконтроллеров семейства CC13XX от Texas Instruments. Для курса мы будем использовать комплект разработчика CC1310 LaunchPad™. Его стоимость составляет порядка 30$ на сайте производителя. Для передачи данных, нам понадобится два таких устройства, один будет выполнять функцию датчика, второй приемника. Ссылка на продукт: https://www.ti.com/tool/LAUNCHXL-CC1310.

Что необходимо знать

Этот учебник предполагает, что у вас есть базовые знания программирования на языке Си. В частности, понимание арифметики указателей, что является распространенной проблемой у молодых программистов.

Чему вы научитесь

В конце курса вы приобретете понимание как создают IoT устройства, а также создадите свой первый IoT проект. Мы создадим систему из центрального устройства и IoT датчика/ов.

Структура учебника

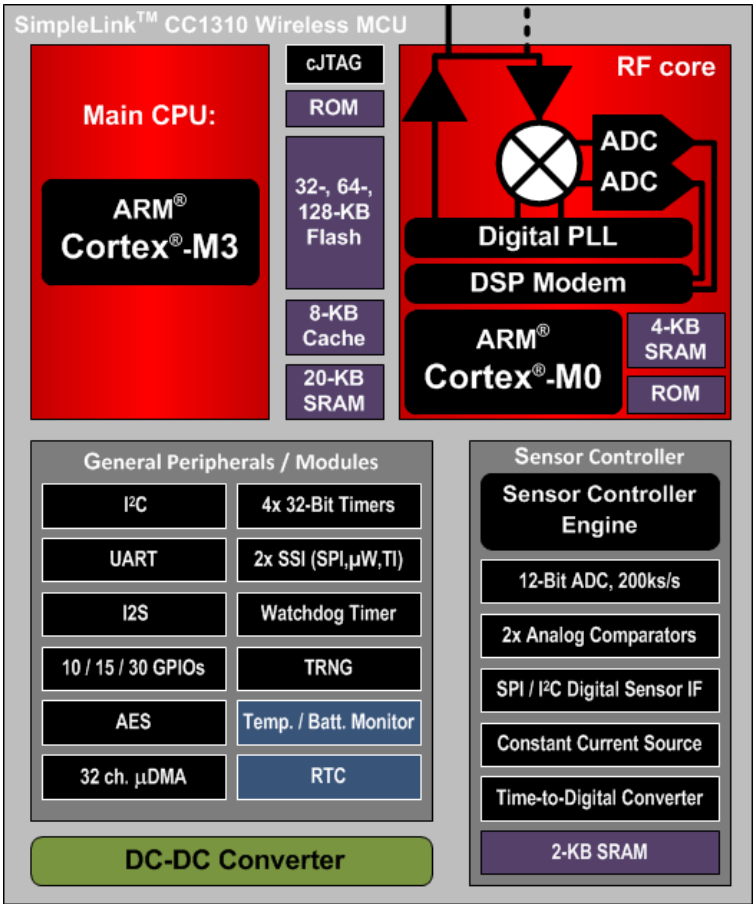
Учебник состоит из уроков. Первые шесть уроков описывают работу основных модулей микроконтроллера, которыми мы будем пользоваться в конечном проектом. В начале урока есть немного теории, для понимания принципа работы того или иного модуля. Остальные уроки — это имплементация проекта.

Микроконтроллер

Микроконтроллер — это микросхема, предназначенная для программного управления электронными схемами. На нем расположено как вычислительное устройство, так и ПЗУ и ОЗУ (в отличие от микропроцессора). Кроме этого, в составе микроконтроллера чаще всего находятся порты ввода/вывода, таймеры, АЦП, последовательные и параллельные интерфейсы. Интересный факт - первый патент на микроконтроллер был выдан в 1971 году компании Texas Instruments. Одно из главных минусов микроконтроллеров – это количество памяти. Обычно оно измеряется в килобайтах, поэтому мы не можем особо раскидываться памятью и должны оптимизировать наш код по максимуму, для того, чтоб уместиться в выделенные нам рамки. С другой стороны, микроконтроллеры в разы дешевле микропроцессоров.

Итак, что же имеет наш процессор? Если мы взглянем на блок схему, то мы увидим следующие модули (рисунок 1):

- Главное вычислительное устройство, основанное на базе Arm® Cortex®-M3 процессора. Этот модуль по суди и исполняет наше приложение (программу)
- ПЗУ (ROM) содержит в себе базовый загрузчик и некоторые встроенные функции
- ПЗУ (Flash) – 32кБ, 64кБ или 128 кБ (в зависимости от версии) памяти, для хранения приложения или данных.
- ОЗУ (SRAM) – память, используемая приложением (20 кБ)
- ОЗУ (GPRAM) – память, которая может использоваться как кеш, либо как обычная память (8 кБ)
- 4 32-битных таймера
- RTC (часы реального времени)
- TRNG (генератор случайных чисел)
- Watchdog (сторожевой таймер)
- Цифровые вводы/выводы (GPIO)
- 12-битный АЦП
- Модуль шифрования AES-128
- Встроенные датчики температуры и напряжения
- 32-канальный DMA
- Интерфейсы I2C, UART, I2S, SPI
- Сверхмало потребляемый автономный контроллер датчиков (2 кБ памяти)


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

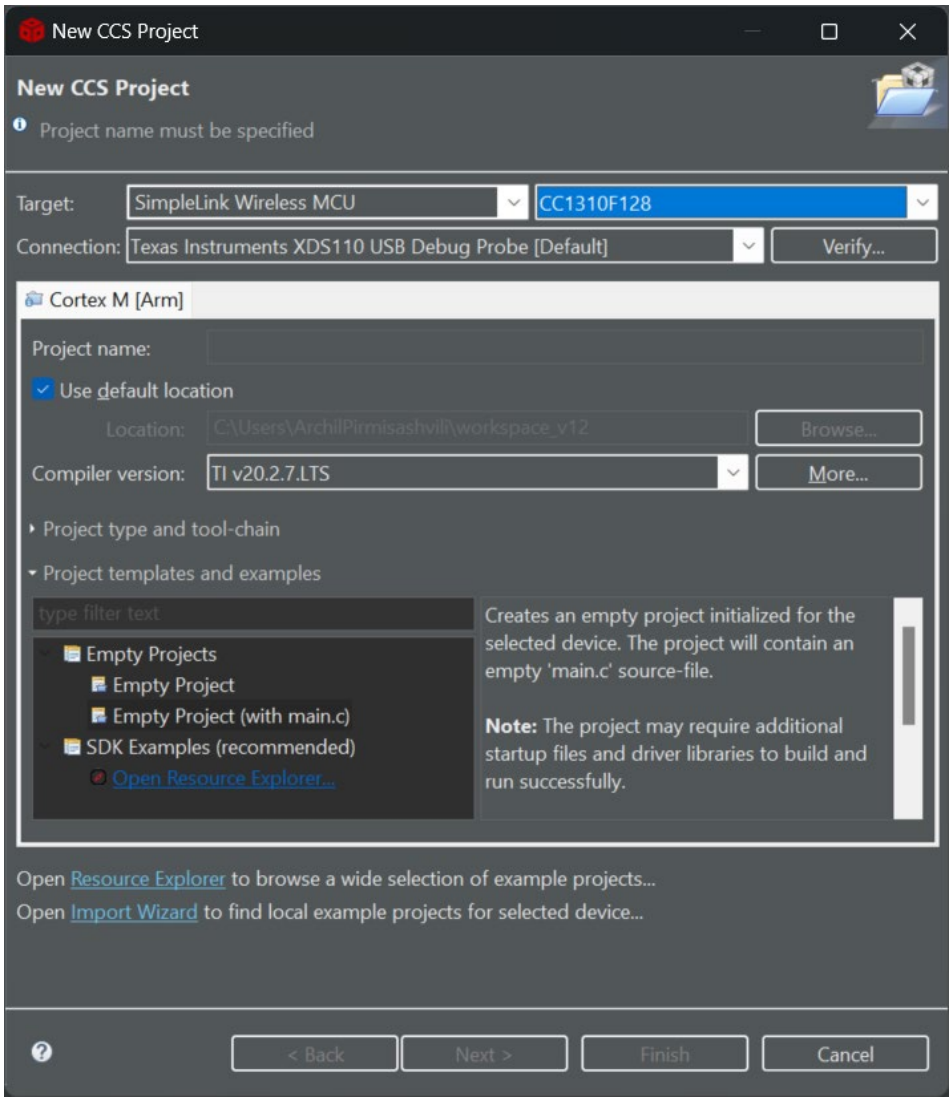
Подготовка рабочей среды

Для разработки программного обеспечения микроконтроллера нам понадобятся следующие инструменты:

- Инструменты генерации кода (компилятор) - https://www.ti.com/tool/ARM-CGT
- Комплект разработки программного обеспечения (SDK) - https://www.ti.com/tool/SIMPLELINK-CC13X0-SDK
- Интегрированная среда разработки - https://www.ti.com/tool/CCSTUDIO
- Приложение для прошивки контроллера (опционально) - https://www.ti.com/tool/FLASH-PROGRAMMER

Создание проекта

Итак, все инструменты установлены и нам уже не терпится начать работу. Начнем с создания проекта. Запускаем среду разработки (Code Composer Studio IDE) и создаем наш первый проект.

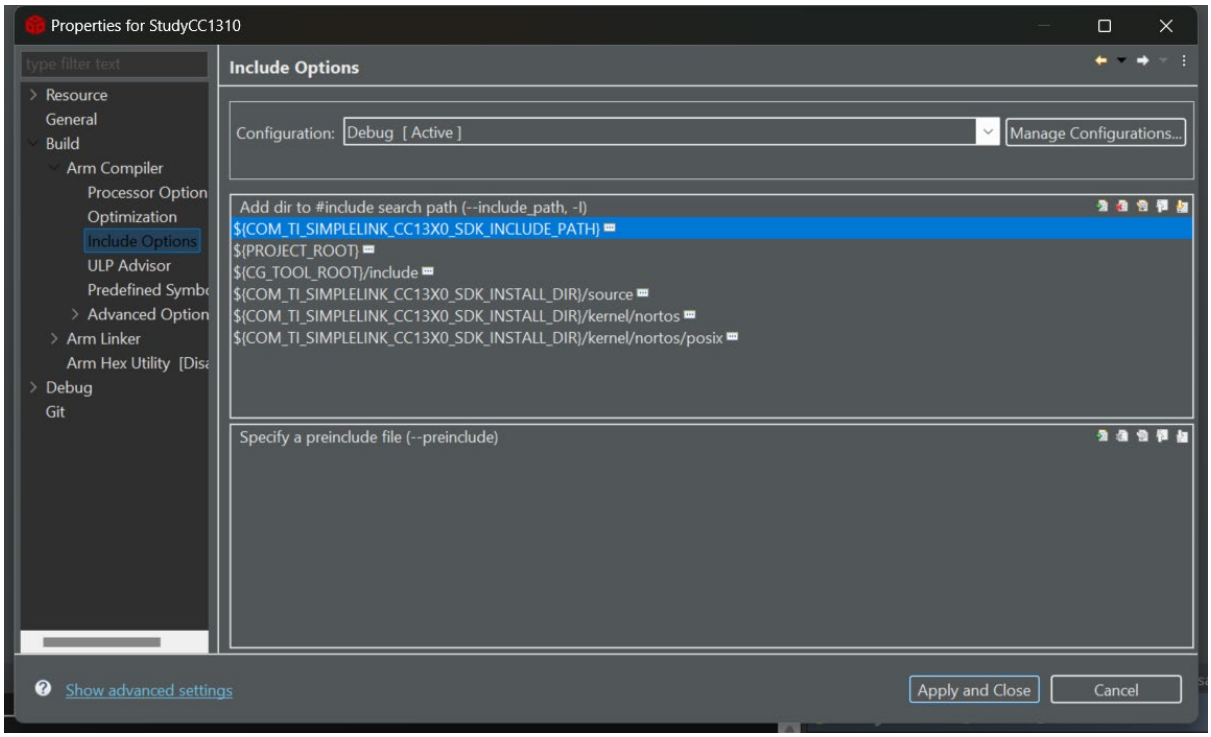


В поле «Target» выбираем «Simplelink Wireless MCU», и модель контроллера «CC1310F128». Создаем проект с файлом main.c.

Далее настраиваем компилятор и линкер. Для этого открываем окно Properties проекта (комбинация кнопок Alt + Enter). В отделе General открываем Products и

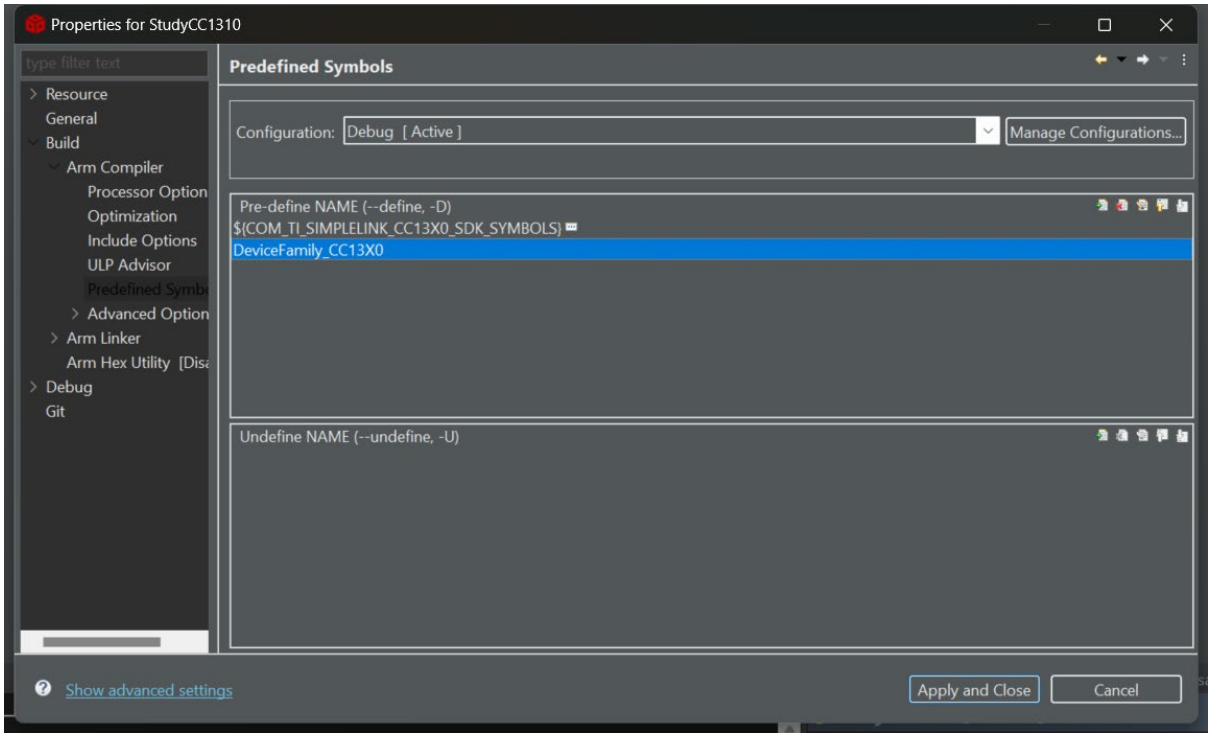
добавляем Simplelink CC13x0 SDK. Далее добавляем пути к SDK для компилятора (Build->Arm Compiler->Include Options):

```
${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
```



Далее в Predefined Symbols (Build->Arm Compiler->Predefined Symbols) добавляем:

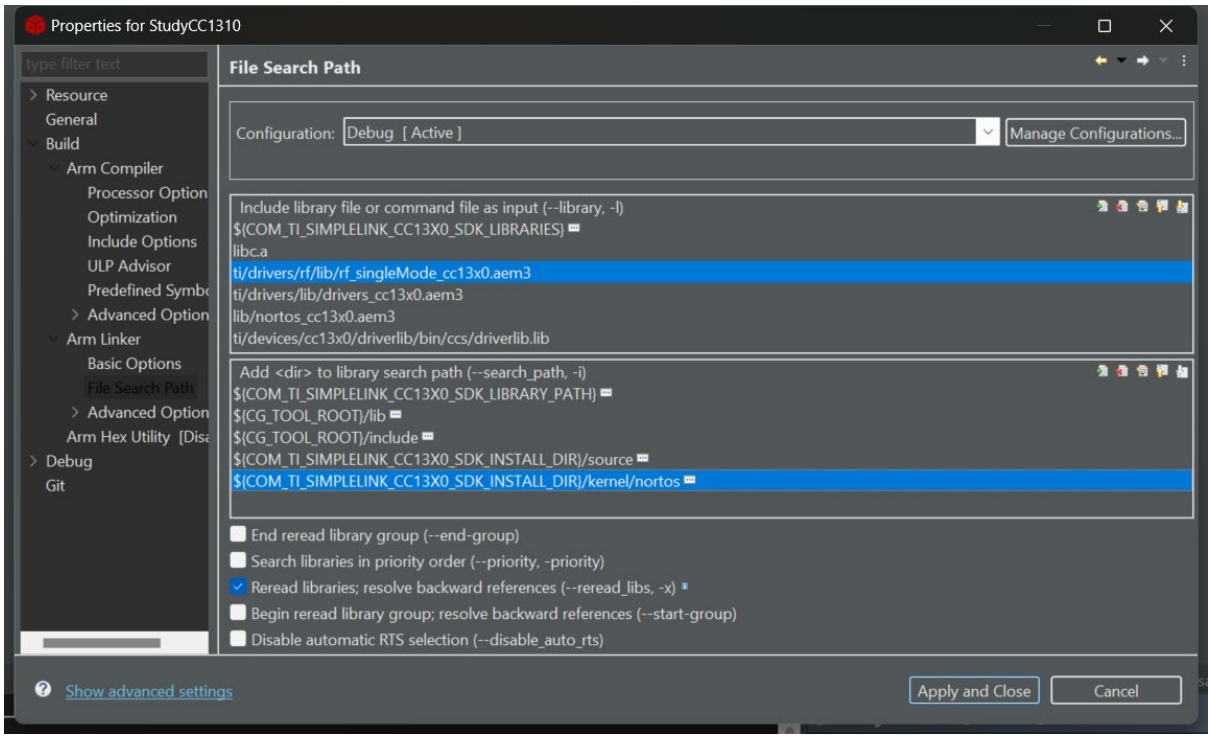
```
DeviceFamily_CC13X0
```



Настраиваем линкер (Build->Arm Linker->File Search Path). Добавляем пути к библиотекам и подключаем их.

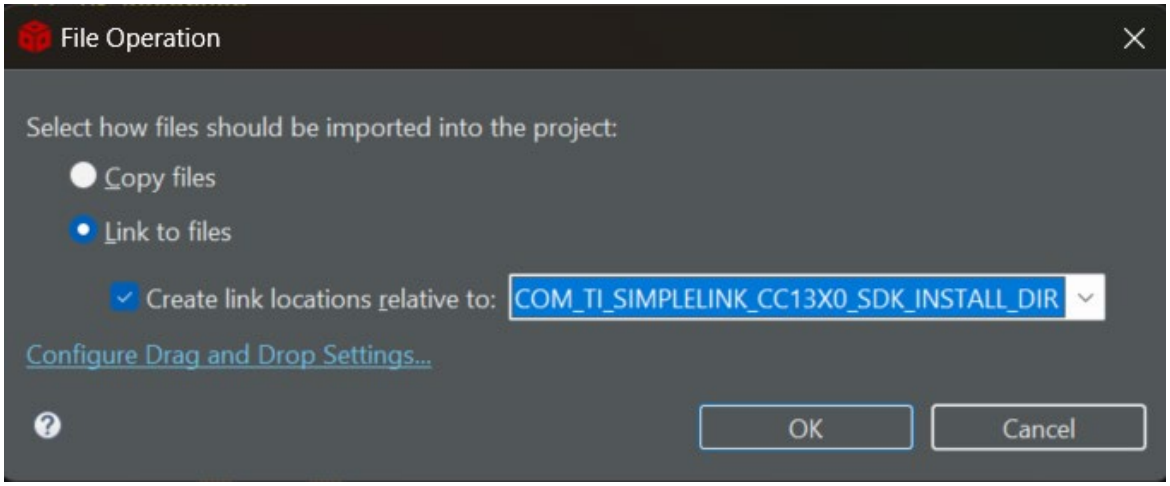
```
ti/drivers/rf/lib/rf_singleMode_cc13x0.aem3
ti/drivers/lib/drivers_cc13x0.aem3
lib/nortos_cc13x0.aem3
ti/devices/cc13x0/driverlib/bin/ccs/driverlib.lib
```

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



Обратите внимание что в проекте есть две конфигурации Debug и Release. Вторая конфигурация оптимизирует код и генерирует исполнительный файл меньшего размера. Поэтому данные настройки стоит проделать в обоих конфигурациях.

Создатели SDK позаботились о нас и приготовили файл, который содержит настройки вектора прерываний микроконтроллера. Перетянем его (startup_ccs.c) в наш проект и добавим, как ссылку (из папки SDK/source/ti/devices/cc13x0/startup_files).



Все, настройка завершена. Теперь наш проект может быть успешно собран.

Особенности программирования микроконтроллеров

Поскольку микроконтроллеры имеют мало памяти, мы должны стараться экономить на «всем чем можно». Поэтому без особой нужды мы не должны пользоваться всеми возможностями современных компьютеров, а именно для того, чтоб «зажечь лампочку» или «послать сообщение» или что-то в этом духе, нет особой надобности использовать операционную систему. Разумеется, существуют операционные системы для микроконтроллеров, но в нашем случаи их использование излишне.
Еще одно важное правило программирования микроконтроллеров – это запрет на использование динамического распределение памяти или по понятное каждому программисту использование функции malloc. Дело в том, что встроенных систем маленькая память и со временем частые вызовы функций выделения/освобождения памяти могут вызвать сильную фрагментацию и утечки памяти. И очень часть использование динамической памяти одна из главных причин багов.

Многие начинающие программисты не понимают значение ключевого слова volatile. А во встроенных системах оно часто используется. Задача этого ключевого слова, предотвратить оптимизацию переменной компилятором. Например, возьмем следующий псевдокод:

```
int i = 0;

void irq_callback() {
    i++;
}

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

Проблема этого кода заключается в том, что переменная i изменяется не в цикле while, а в обработчике прерывания, другими словами, из другого потока. Компилятор поймет, что переменная не меняет значения и условие цикла всегда выполняется, значит можно ее оптимизировать в следующий код:

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

Довольно часто компиляторы не оптимизируют такую переменную, но стоит нам поднять уровень оптимизации, как мы получим неисправный код. Потом тратим кучу времени чтобы найти причину неисправности. Именно поэтому важно добавлять ключевое слово volatile.

Структура проекта

Для простоты использования и понимания кода, поделим проект по папкам. Каждая папка будет содержать файлы, с определенными частями/модулями.

Boards – в этой папке будем хранить файлы связанные с конкретным устройством. В этом учебнике мы будем использовать только одно устройство, но, если вы решите добавить новое, это не доставит вам особого труда.

Drivers – в этой папке мы будем хранить драйверы. В основном драйверы периферийных устройств/интерфейсов.

Apps – в этой папке мы будем хранить файлы приложений, которые мы будем запускать на устройстве.

Modules – в этой папке мы будем хранить файлы модулей конечного проекта.

Smartrf_settings - в этой папке мы будем хранить файлы настройки радио.

Tests – в этой папке мы будем хранить файлы тестов, который мы будем использовать для проверки функциональности нашего кода.

Tools – в этой папке мы будем хранить вспомогательные файлы, такие как командные файлы.

Файл ti-lib.h содержит в себе вспомогательные макросы и заголовки библиотек разработчика

SDK

В этом учебнике мы будем использовать Simplelink CC13X0 SDK. Эта библиотека предлагает нам удобный интерфейс работы с микроконтроллером. По сути, работа с любым микроконтроллером заключается в работе с его регистрами, изменяя значения которых, мы можем использовать

модули/периферию микроконтроллера. Работа с регистрами не особо удобна обычному человеку, для этого разработчик микроконтроллера предлагает нам библиотеку функций понятную обычному программисту.

Один из основных модулей библиотеки – NoRTOS. Данный модуль предоставляет нам возможность работы с микроконтроллером без операционной системы. Он позволяет нам использовать задержки (delay), берет на себя регистрацию прерываний, системные часы, управление энергосбережением и многое другое.

Исходный код

Исходный код вы можете найти по адресу:
https://github.com/Kulipator/DIY-IoT-Programming
Каждая часть находится в отдельной ветке.

Архитектура Superloop

Архитектура микроконтроллера, равно как и микропроцессора, построена так, что с момента подачи питания на микроконтроллер, иначе говоря включения, он должен постоянно выполнять команды. Даже в случае простоя, он выполняет так называемую команду NOP. В случае с операционными системами последняя берет на себя эту ответственность, другими словами, когда нет никакой запущенной программы, операционная система запускает IDLE процесс (процесс простоя). В нашем же случае, поскольку у нас нет операционной системы, наша программа не должна никогда завершаться. Иначе говоря, наша программа должна иметь бесконечный цикл. Отсюда и название архитектуры.

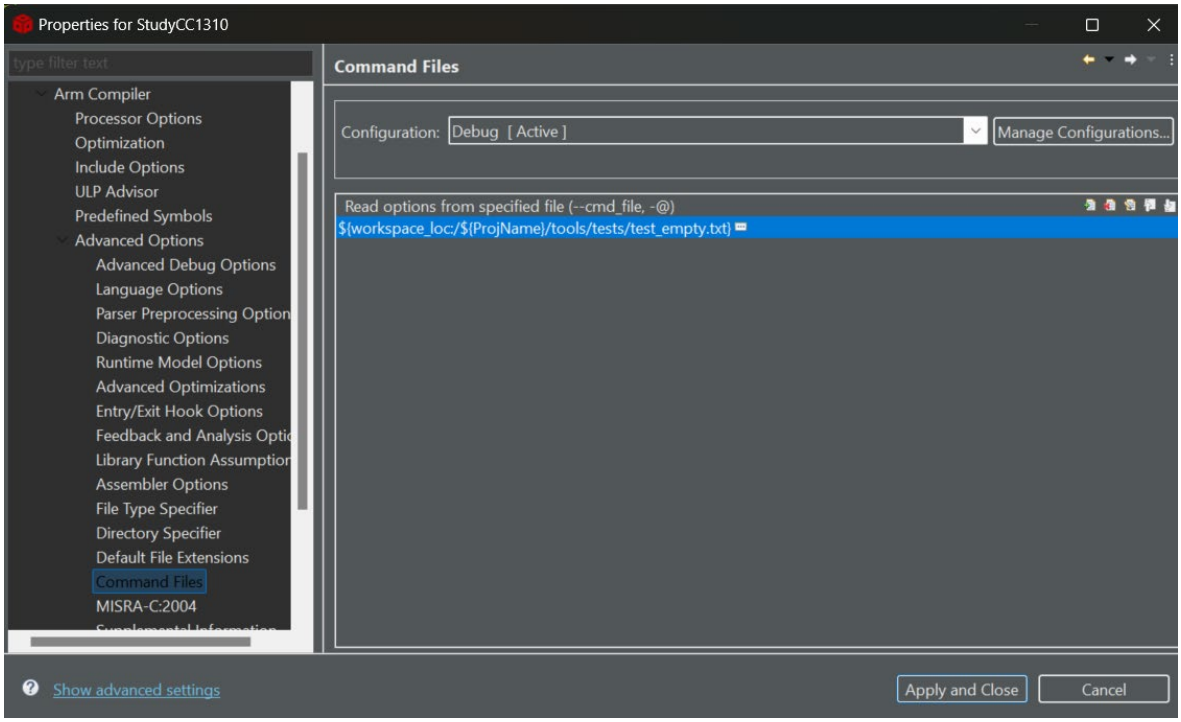
Эта архитектура подразумевает особую концепцию программирования - асинхронную. Возьмем пример, у нас есть система, которая должна обрабатывать ввод пользователя и посылать данные по радиоканалу и наоборот, принимать данные с радиоканала и отправлять их на вывод пользователю. Самое простое решение этой задачи – ждать ввод пользователя, и по получению ввода отправить данные, но что произойдет если в этот момент придут данные с радиоканала? Мы не сможем их обработать, потому что заняты приемом данных от пользователя, и только после того, как мы получим ввод пользователя, мы сможем обработать данные с радиоканала. Это решение нам не подходит, так как приводит к задержкам, а в случае большого количества данных – к переполнению буферов и потере данных. Следовательно функция считывания ввода пользователя не должна ждать окончания ввода, а возвращаться сразу, к примеру проверяет есть ли данные и в случае, если они имеются обработать их, иначе выйти.

Программирование микроконтроллера

Первое приложение

Первым приложением создадим пустое приложение, которое инициализирует наше устройство и больше ничего не делает. По ходу продвижения мы будем дополнять его.
Начальная точка входа находится в файле main.c, функция main. Давайте посмотрим, что она делает? Первым делом мы должны инициализировать периферию нашего устройства с помощью вызова функции board_initialize. Далее инициализируем модуль NoRTOS библиотеки, и передаем управление нашему приложению, вызвав его функцию запуска (mainThread). На случай если мы по ошибке выйдем из главной функции приложения, добавим бесконечный цикл, который не даст, завершится нашей программе. Этот цикл играет роль защиты, если мы все делаем корректно, то мы никогда не достигнем этого цикла.

Поскольку это приложение ничего не делает, сочтем его за тест. Следовательно, разместим главный файл приложения в папке Tests. Если вы обратите внимание, то в этом файле код заключен в директиву препроцессора #idef, этот подход нам дает возможность иметь множество файлов с функцией mainThread и использовать нужный нам файл с помощью директивы препроцессора. Данную директиву мы разместим в файле команд в папке tools. На стадии компиляции нам нужно будет выбрать нужный нам файл команд, и компилятор нам соберет нужное приложение. Чтобы выбрать файл команд откройте настройки проекта и выберите нужный файл в отделе Build->Arm Compiler->Advanced Options->Command Files.



Вы можете видеть, что в файле test_empty.txt есть два определения: устройство (BOARD) и приложение.

Файл приложение empty_test.c, по сути, имеет бесконечный цикл в главной функции. Тем самым приложение ничего не делает.

Теперь рассмотрим папку boards. В ней у нас находятся файл board_common, который содержит функции общие для всех устройств, к примеру функция инициализации, а также прототип функции входа в приложение (mainThread).

Также, вы можете увидеть, что в зависимости от определения BOARD, подключаются нужные файлы из папки устройства.

Попробуем собрать наше первое приложение. Для этого нажмем комбинацию клавиш CTRL + B, либо кликнем на иконку с рисунком молотка. А далее мы можем запустить приложение в режиме отладки при помощи нажатия иконки с рисунком жучка. Viola, наша первая программа запущена на устройстве, но мы не видим никаких признаков ее работы, так как она ничего не делает.

Да будет свет!!!

Наше первое приложение ничего не делало и было не понятно работает оно или нет. Добавим ему немного интеракции. Самый простой способ — это зажечь лампочку (LED), их у нас на устройстве имеется даже целых две. Но сначала немного теории и электроники.

Все мы знаем, что для того, чтоб зажечь лампочку надо пустить через нее ток. Но как это сделать на нашем устройстве? По сути, лампочка должна иметь два состояния, включенное или выключенное. Для контроля этих состояний нам идеально подходит модуль цифрового ввода/вывода (GPIO).

Так как устроен модуль GPIO? Этот модуль один из основных в каждом микроконтроллере. Имеется какое-то количество контактов, которые могут выполнять функции GPIO. Каждый такой ввод/вывод имеет номер и может работать в 2 состояниях (либо ввод, либо вывод), другими словами, принимать состояние 0 или 1. Но как контроллер знает где 1, а где 0. Для этого попробуем понять, как работает цифровая электроника. Для работы любого устройства необходим источник питания, он питает электрическую цепь посредством электрического тока. Но как ток может превратиться в цифру? Все очень просто, если есть ток – то это логическая единица, нет – 0. Но на самом деле не совсем все так. Дело в том, что за логическую единицу принято считать промежуток напряжения близкий к верхней границе напряжения источника питания (в электронике 3.3 вольта либо 5 вольт, в зависимости от системы), а логический 0 – промежуток напряжения близкий к 0. В случае с лампочкой, если мы будем использовать цифровой выход устройства и переключим его состояние в 1, то на его выходе появится напряжение и если к нему подключить лампочку, то она загорится.

Давайте рассмотрим режимы GPIO. Как мы уже поняли существуют два режима: ввод или вывод. Каждый из них имеет свои режимы.

Режимы ввода

Обычный (No pull) – этот режим ввода используется по умолчанию в микроконтроллерах. Входное напряжение конвертируется в логическое значение и выдается как результат измерения. Но тут есть проблема. Что будет если входное напряжение будет где-то посередине между 0 и 1? Для решения такой ситуации нам нужно «подтягивать» напряжение в одну из сторон. Это можно решить с помощью подтягивающего сопротивления. Это сопротивление можно поставить как физический компонент, либо использовать встроенный режим подтяжки GPIO.

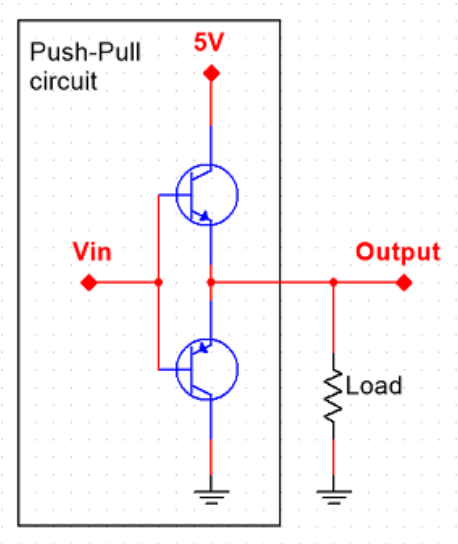
Подтяжка вверх (Pull up) - этот режим ввода используется в случае, когда цепь принимает значение 0, либо неизвестное. Классический пример – кнопка/переключатель. Если цепь построена так, что переключатель соединяет вход с землей, то в случае, когда переключатель разомкнут, на входе образуется

неизвестное значение (либо 0) в таком случае мы не сможем понять кнопка нажата или нет. Но если мы будем «подтягивать» напряжение вверх, то в случае разомкнутого переключателя мы получим 1 на входе, а в случае замкнутого – 0.

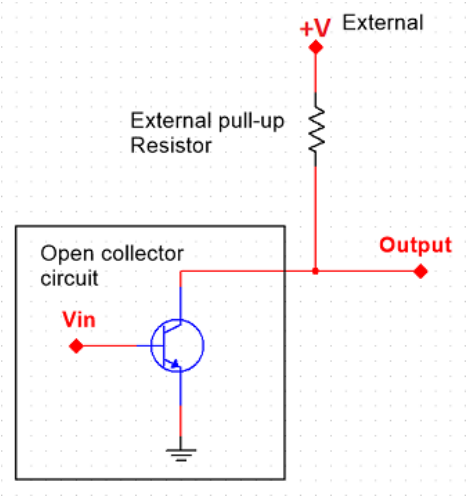
Подтяжка вниз (Pull down) – обратный режим от предыдущего. В нем мы «подтягиваем» напряжение вниз. Как пример можем использовать тот же переключатель только замыкающий вход с +.

Режимы вывода

PushPull – в этом режиме вывод принимает значение либо 1, либо 0. Другими словами, оно не может быть неизвестным, так как либо подтягивается вверх, либо вниз. Отсюда и название.



OpenDrain – в этом режиме вывод принимает значение либо 0, либо неизвестное. По сути, этот режим используют различные интерфейсы передачи данных, такие как I2C, когда на шине имеются несколько устройств и для того, чтоб не мешать работе других устройств. Чтобы понять зачем нужен этот режим, то попробуем соединить два вывода PushPull и поставим их в разные состояния. Из-за конфликта уровней (потенциалов) у нас пойдет ток в цепи и один из выходов может выгореть.



OpenSource – этот режим похож на предыдущий с разницей в том что вместо 0 получает значение 1.

С теорией разобрались, перейдем к практике. Наша задача управлять LED-ами. Тут имеет смысл создать драйвер, целью которого будет управление LED-ами. Драйвер будет поддерживать до 4 LED-ов и сможет управлять ими. Нам нужно уметь зажечь, потушить и изменить состояние LED-а. Еще одна функция – привязать к мультиплексору (мы ее будем использовать для индикации приема/передачи по радио).

Файлы драйвера будут находиться в папке drivers.

Код инициализации устройства будет инициализировать драйвер, так как этот драйвер нам понадобится во всех случаях чтобы давать индикацию что что-то происходит. Поэтому имеет смысл добавить его в код инициализации, а не инициализировать в каждом приложении что мы будем писать. Для управления LED-ами мы будем использовать модуль PIN библиотеки. Для использования этого модуля мы должны подключить нужные заголовки и функции, добавим их в файл ti-lib.h.

```
 /*---------------------------------------------------------------------------*/
 /* 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__)
```

Также для использования этого модуля нужно создать константы (PINCC26XX Hardware attributes и PIN_Config). Поскольку каждое устройство может иметь разные входы/выходы, которые нужно инициализировать при запуске, объект PIN_Config будет находится в файле устройства. В нашем случае это BoardGpioInitTable который пустой.

```
/*
 *  ================================= 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
};
```

Наше приложение будет зажигать и тушить LED-ы с разной частотой. Для этого нам также понадобится функция задержки (delay), которая находится в модуле NoRTOS. Эта функция переводит устройство в спящий режим во время ожидания, поэтому нам также понадобится модуль управления

энергосбережением. Для него так же, как и для модуля PIN следует подключить заголовки и добавить нужные константы (PowerCC26XX_config).

Теперь инициализация устройства будет выглядеть следующим образом: инициализация энергосбережения, инициализация модуля PIN (GPIO) и инициализация драйвера LEDs.

```
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();
}
```

В драйвере LEDs мы создаем массив LED-ов (driver_leds_leds), а также массив конфигурации выводов (driver_leds_pin_table). Каждый вывод мы настраиваем как вывод в конфигурации PushPull (понятно почему, мы ведь хотим получить значение 0 либо 1) с начальным погашенным состоянием (PIN_GPIO_LOW). Далее функция инициализации инициализирует выводы и ставит их в начальное состояние.

Функции управления LED-ами меняют состояние выбранного вывода в нужное положение. Стоит обратить внимание на функцию toggle (смена состояния), мы читаем состояние вывода и записываем обратное.

Также нам понадобятся задержки между изменением статуса LED-ов. Их можно использовать различными способами. В связи с тем, что микроконтроллеры часто питаются от обычных батарей, нам следует экономить энергию и в периоды, когда микроконтроллер ничем ни занят, или определенный модуль микроконтроллера не используется, следует его отключить, другими словами – отправить в спящий режим. Задержка – это идеальный пример такой ситуации. Добавим следующие макросы в файл board_common.h:

```
#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(); }
```

Данные макросы переводят микроконтроллер в спящий режим на заданный период времени. Их мы и будем использовать в промежутках между изменением статуса LED-ов.

Соберем приложение и запустим его. Мы видим, что LED-ы зажигаются и гаснут с частотой от 100 миллисекунд и до 5 секунд с шагом 100 миллисекунд.

Сторож (Watchdog)

Во встраиваемых системах часто происходят сбои, которые могут привести к «зависанию» системы. Для таких случаев создан механизм сторожа. Смысл его заключается в том, что программа сообщает сторожу что она функционирует. Это выполняется путем изменения логического значения ее контакта в случае аппаратного решения, или вызова функции сброса на программном уровне. Если изменение не произошло в заданный период времени, то сторож перезапускает систему.

Создадим драйвер сторожа. Мы будем использовать модуль Watchdog библиотеки.

Добавим необходимые заголовки в файл ti-lib.h.

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

В драйвере создадим необходимые константы:

```
/*
 *  ============================== 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;
```

Как вы видите, мы настраиваем период времени на 1 секунду. Это значит что, если в течении секунды сторож не был оповещен, он перезапустит систему.

Также добавим инициализацию сторожа в код инициализации устройства.

```
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();
}
```

Стоит заметить, что при переходе устройства в «спящий режим», сторож продолжает работу. Поэтому следует приостановить перезапуск системы во время «сна». Для этого мы будем использовать флаг (watchdog_disabled). Добавим отключение сторожа в макро спящего режима.

```
#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); }
```

Также в отладочном режиме если мы остановим программу, то сторож продолжит работать, что приведет к обрыву отладочного процесса. На этот случай предусмотрен режим сторожа, который также останавливает его. Для его включения или отключения будем использовать директиву препроцессора WATCHDOG_STALL во время инициализации сторожа.

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

Так же для удобства создадим файл (common_utilities) вспомогательных функций, такие как время с запуска системы (get_up_time, get_up_time_us), или программный перезапуск (reboot).

Добавим необходимые заголовки в файл ti-lib.h.

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

Обратите внимание на следующий код:

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

Этот код гарантирует нам атомарное выполнение кода заключенного внутри. Атомарность осуществляется отменой прерываний. Поэтому важно производить минимум операций внутри атомарного блока, чтоб не пропустить прерывание.

Для получения времени с запуска системы мы используем модуль RTC (aon_rtc в библиотеке). По сути, это таймер, который запускается с запуском системы.

Чтобы проверить функционал сторожа, создадим тестовое приложение, которое основывается на предыдущем тесте с той лишь разницей, что после обнуления времени «спячки» приложение перестанет оповещать сторож и тем самым приведет к перезапуску системы. Важно понимать, что сторож работает даже в то время, когда микроконтроллер находится в спящем режиме. Поэтому мы должны отключать перезапуск системы на время «сна». Для этого изменим макросы сна.

```
#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(); }
```

Как вы видите в файле watchdog_test.c, мы создали переменную stop_watchdog, которая будет выполнять функцию флага. Пока он не поднят, мы обнуляем счетчик сторожа с помощью вызова функции WDT_RESET(). При его поднятии мы перестаем обнулять счетчик сторожа.

Важный момент! При подключенном отладчике (debugger), а также, при вызове сторожа в момент, когда устройство находится в спящем режиме, устройство будет зависать. Для отключения отладчика следует убрать перемычки (jumpers) TMS, TCK, TDI, TDO, SWO.

Серийный ввод вывод - универсальный асинхронный приёмопередатчик (UART)

Один из способов обмена информации это UART. Это серийный протокол, который имеет пару линий информаций, для её передачи. В связи с тем, что имеется две линии, информацию можно передавать в обе стороны одновременно (асинхронно). Информация передается по битам за равные промежутки времени, который определяется скоростью передачи. Приемник обозначается RX, а передатчик TX. Соответственно два устройства подключаются друг к другу RX к TXу.
Мы будем использовать UART в качестве консоли/терминала приложений. Для этого создадим драйвер Console.

Добавим необходимые заголовки в файл ti-lib.h.

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

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

А также создадим необходимые константы:

```
/**
 *  @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;
```

В файл устройства добавим объявление контактов для интерфейса UART.

```
/*----------------------------------------------------------------------------*/
/**
 * @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 */
/** @} */
```

Драйвер имеет два буфера, один для приема другой для передачи. Они будут использоваться для временного хранения информации. Главный цикл должен вызывать функцию console_process для обработки буферов и запуска передачи в нужных случаях.

При чтении данных, мы используем буфер, потому что информация может приходить по частям. И нам нужно как-то различать начало конец сообщения. Существуют несколько способов это делать. Один из них – это разделять сообщения промежутком времени. К примеру, сообщения не могут приходить с периодом меньшим чем 100 миллисекунд. Другой способ обозначать начало и/или конец сообщения специальным символом. К примеру – сообщение оканчивается символом «CR» (Carriage Return). Для этого будем использовать следующие директивы препроцессора:

CONSOLE_RX_TIMEOUT_MS – Таймаут в миллисекундах между двумя последующими сообщениями.

CONSOLE_END_CHAR – Символ окончания сообщения.

В случае если CONSOLE_END_CHAR не определен, то будет использоваться CONSOLE_RX_TIMEOUT_MS. Для того чтоб изменить CONSOLE_RX_TIMEOUT_MS с командного файла, следует задать директиву препроцессора CONSOLE_CONF_RX_TIMEOUT_MS.

При вызове функции записи драйвера, информация копируется в буфер и запускается передача. Если во время записи передача была в процессе, то по окончанию передачи, драйвер проверит, есть ли данные в буфере отправки и если он не пустой, то начнет новую передачу. Согласно документации производителя, функции UART_read и UART_write не могут вызываться из своих функций обратного вызова (callback), поэтому они вызываются из функции console_process.

При получении входящей информации, вызывается обратная функция чтения, которая регистрируется при помощи функции console_register_message_received_callback.

Для проверки драйвера создадим тест console_test, который будет выполнять функцию эхо. Это значит, что устройство будет возвращать ту же информацию, которую будет получать. А также для индикации приема/передачи будет использовать зеленый LED (с помощью директивы препроцессора CONSOLE_LED=LEDS_GREEN). В этом примере используется «CR» для разделения сообщений, поэтому следует посылать сообщение, оканчивающее на этот символ. Откройте терминал (Docklight, Putty или любое другое приложение) и подключитесь к серийному порту с названием «XDS110 Class Application/User UART» со следующими параметрами:

BaudRate: 19200

Parity: None

DataBits: 8

StopBits: 1

Работа с памятью FLASH

Очень часто нам требуется в приложении изменять и сохранять настройки. Для этого нам нужен доступ к постоянной памяти. В микроконтроллерах для этой цели используют FLASH память. Важно отметить, что сама программа хранится на этой же памяти, поэтому важно использовать ту часть памяти, где не находится приложение. Обычно в таких целях используют конечную часть памяти, чтоб избежать накладывания программы на настройки.

Структура FLASH памяти такова, что она поделена на страницы одинакового размера. В микроконтроллерах CC1310 размер страницы составляет 4кБ. Если объем памяти микроконтроллера составляет 120кБ, то у нас имеется 32 страницы памяти. Изначальное состояние битов памяти – 1, и мы можем его изменить на 0 или оставить 1. Но если мы изменили бит памяти на 0 и хотим вернуть его обратно на 1, то это сделать невозможно. Единственный способ вернуть бит памяти с 0 в 1 – это очистить страницу. В таком случае мы потеряем все остальные состояние битов памяти на этой странице. Также FLASH память имеет ограниченное количество стираний. Другими словами, что после определенного количества очисток страниц, память может работать не корректно. Обычно это количество составляет десятки тысяч. Для большей жизнедеятельности FLASH памяти, следует использовать очистку страниц помнимому, и применять различные алгоритмы для сохранения нужных нам данных. В микроконтроллерах CC1310 последние 88 байтов памяти используются как так называемые регистры настройки, которые задают параметры работы микроконтроллера. В данном курсе мы не будем их затрагивать.

Рассмотрим драйвер FLASH памяти. Он имеет следующие функции:

flash_load – эта функция читает определенное количество байтов из FLASH памяти. По сути чтение памяти в микроконтроллере это чтение с определенного адреса. Вы можете видеть, что мы используем функцию memcpy для этих целей.

flash_erase_page – эта функция очищает страницу памяти. Как параметр она получает номер страницы. Обратите внимание что во время стирания страницы следует отменить прерывания, это вы можете видеть, что вызов функции очистки страницы помещен в атомарный блок:

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

Так же в микроконтроллерах CC1310 часть памяти можно использовать для кеша. В таком случае, следует отключить кеш перед очисткой страницы, а потом включить ее обратно, что мы и делаем:

```
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 – эта функция записывает некое количество байт в память. Так же, как и в предыдущей функции, операция записи должна производиться с отключенными прерываниями и кеш памятью. Обратите внимание что она не очищает страницу, поэтому перед тем, как записывать данные, стоит убедиться, что страница очищена.

flash_load_ccfg – эта функция считывает данные регистров настройки.

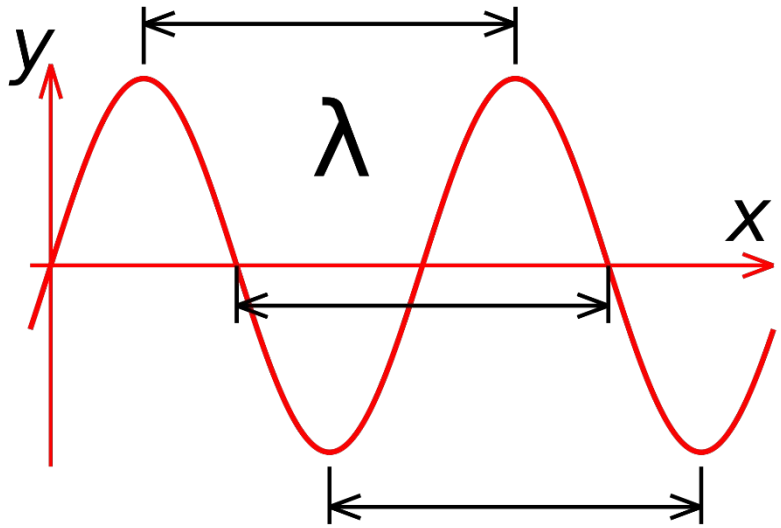
flash_save_ccfg – эта функция записывает регистры настройки.

Для проверки драйвера мы создадим тест, который будет использовать страницу 30. Первый шаг – это очистка страницы, затем, запись данный, и последний шаг чтение данных и сравнение их с первоначальными. Результаты каждого шага будем выводить на консоль, используя драйвер из предыдущего примера. Поскольку наш тест имеет несколько состояний, будем использовать автомат состояний. Для этого используем переменную step, которая будет показывать текущее состояние. При достижении состояния автомат останется в этом состоянии до перезапуска.

Так же во встроенных системах принято при старте как-то оповещать пользователя об этом. Самое простое решение — это использовать LED. При запуске программы мы поморгаем всеми LED-ами 2 раза. Для этого воспользуемся функцией **leds_blink_all** драйвера LED-ов.

Радиосвязь

Для того чтобы передавать данные по радио, следует понять, как вообще работает этот процесс. Для начала вспомним что такое радиоволны и как мы можем их использовать для решения нашей задачи. Из курса физики нам известно, что радиоволны – это электромагнитные колебания. Также известно, что электромагнитные волны распространяются в вакууме со скоростью света. Молния является естественным источником радиоволн. Искусственно созданные радиоволны используются для связи, а в нашем случае, передачи данных «по воздуху». Для передачи и приёма данных используются приёмник и передатчик. Часто эти два устройства находятся в одном, который может посылать и принимать данные. Если мы посмотрим на волну, то мы увидим:



У волны есть 3 основных свойства:

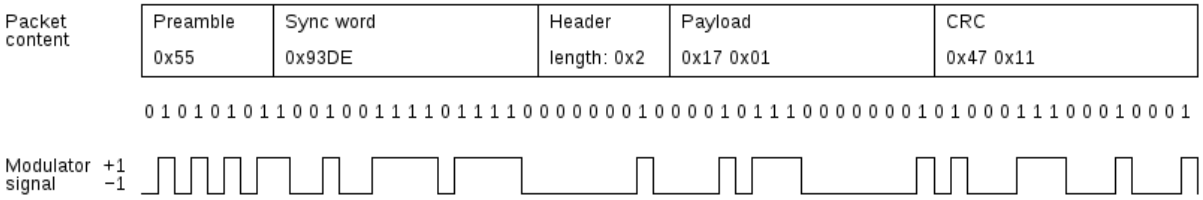
- **Амплитуда** – высота волны
- **Частота** – длина волны (количество колебаний в единицу времени)
- **Фаза** – аргумент синусоидальной функции, отсчитываемый от точки перехода минусового значения к положительному и обратно

Используя эти свойства, мы сможем передавать данные. Так как же эти свойства используются? Всем нам известно, что «мозг» компьютеров и других «умных» устройств, использует двоичную логику. Другими словами, все построено на нулях и единицах. Вся информация представлена в этом же формате. Следовательно, чтоб передать информацию, по сути, нам нужно передать единицы и нули. Используя одну из свойств радиоволны, мы сможем это сделать. Чтоб сильно не уходить в теорию, рассмотрим все на примере. Для примера возьмем свойство, которое мы будем использовать и в последствии, а именно частоту, то же самое можно сделать с любым другим свойством. Другими словами, если мы обозначим одну частоту как 0, а другую как 1, то меняя частоту, мы сможем передавать данные на расстояния. Этот способ

называется «частотная модуляция», на английском Frequency Modulation или FM. Эти две буквы наверняка вам знакомы если вы хоть раз пользовались радиоприемником. Самый простой и понятный «пример из жизни» частотной модуляции – это азбука Морзе. Она состоит из коротких (точка) и длинных (тире) сигналов. Другими словами, сигналы разных частот.

Вернемся в мир IoT. У нас имеется два устройства, которое способно принимать или посылать данные по радиоканалу. Для успешной передачи данных нам нужно настроить их так чтоб они использовали одинаковые настройки, иначе наши старания уйдут в никуда. Один из основных параметров – это основная частота. Допустим мы используем частоту 433.92 MHz. Это значит, что выбранные нами частоты для обозначения нуля и единицы будут находиться по обе стороны от центральной. Допустим 0 – это частота 433.90 MHz, а для 1 - 433.94 MHz. Это отклонение от центральной частоты, называется «Frequency Deviation». Поскольку мы передаем данные бит за битом, другими словами, последовательно, обе стороны должны использовать одинаковую скорость передачи/приема данных. Скорость передачи измеряется в бодах (baud). И последний параметр — это мощность передачи, которая измеряется в dBm. Для того чтоб понять на примере мощность передачи, представьте, что вы разговариваете с другом, и постепенно он от вас отдаляется. Чем дальше вы отдаляетесь, тем громче вам надо говорить, чтоб слышать друг друга.

Сами данные передаются с помощью пакетов размером до 255 байтов. Важно отметить, что чем длиннее сообщение, тем больше времени требуется на его передачу, а это увеличивает шанс того, что во время передачи, какое ни будь другое устройство начнет передачу, тем самым создаст помехи, что может привести к потери данных. Поэтому желательно передавать минимальные по возможности пакеты. Каждый пакет имеет следующую структуру:



- Преамбула – последовательность 0101… или 1010… используется для обозначения начала передачи
- Sync word – последовательность байтов (до 4 байтов) для обозначения начала сообщения. Приемник использует преамбулу для распознания начала передачи и не всегда полностью получает ее, для этого существует Sync word, чтоб знать наверняка, где начало сообщения. Так же это поле можно использовать для разделения разных систем давая каждой другой Sync word.
- Заголовок – в основном содержит размер сообщения
- Payload – данные, которые мы передаем

- CRC – контрольная сумма, для верификации полученных данных и выявлении ошибок

Так же важно отметить, что в разных странах есть ограничения на использования определенных частот. Поэтому важно проверить какую частоту вы будете использовать, чтоб не нарушать закон. За распределение частот отвечает министерство связи. В большинстве стран частоты диапазона 433 MHz доступны для общего пользования.

Думаю, что теории достаточно чтоб понимать основные принципы передачи данных по радио. Более подробную информацию можно легко найти в просторах интернета. Мы же перейдем к делу, а другими словами, разберем драйвер радио.

Если мы посмотрим на блок схему нашего микроконтроллера, то увидим, что за радио отвечает отдельный контроллер (Cortex M0). Он связан с основным контроллером посредством механизма, называемым «дверной звонок». Смысл его заключается в том, что главный контроллер посылает команды радио-контроллеру, а тот при завершении операции, уведомляет центральный посредством прерывания, как бы звонит в дверной звонок. Радио контроллер поддерживает определённый набор команд, используя которые мы можем воспроизводить прием, передачу и другие операции. Так же он имеет очень точные часы частотой 4 мегагерца. Эти часы можно использовать для различных задач, в которых требуется повышенная точность по времени.

Для начала рассмотрим файл smartrf_settings. Этот файл содержит в себе описания команд, которые будет использовать радио-контроллер. Рассмотрим каждую команду:

RF_cmdPropRadioDivSetup – эта команда используется для настройки радио. Она содержит в себе такую информацию, как основная частота, отклонение, мощность, скорость передачи данных.

RF_cmdFs - эта команда используется для настройки синтезатора радиосигнала. Он используется для генерации сигнала, поэтому нужно задать ему рабочую частоту.

cmdRx – команда приёма сообщения. С ее помощью мы задаем параметры и структура пакета во время приема.

cmdTx – команда передачи сообщения. С ее помощью мы задаем параметры и структура пакета во время передачи.

RF_cmdRxTest – команда проверки уровня силы сигнала во время приема. С ее помощью мы можем проверять состояние эфира, и определять уровень шума.

RF_cmdTxTest – команда для генерации сигнала. С ее помощью мы можем генерировать постоянный сигнал «тон» определенной частоты, для калибровки частоты передатчика.

cmdPropCs – команда распознавания передачи. С ее помощью мы можем определять начало передачи. К примеру, нам нужно послать пакет данных, но мы не хотим создавать помех другим устройствам. Поэтому перед началом отправки сообщения можно проверить есть ли устройство, которое в данный момент передает сообщение, и в случае, если эфир пустой, можно отправлять пакет. Этот прием называется «Listen before talk» или LBT.

cmdCountBranch – команда «счетчика ответвления». Она используется в совокупности с предыдущей командой, для решения LBT. По сути, это счетчик, который уменьшается при каждой итерации, до тех пор, пока не достигнет 0.

cmdNop – команда, которая ничего не делает. Она используется для режима ожидания.

RF_cmdPropRxAdvSniff – команда приема с «принюхиванием». Смысл ее заключается в том, чтобы сперва распознать «пронюхать» преамбулу, и в случае ее распознания продолжить прием, иначе остановиться. Этот способ позволяет сэкономить потребляемую энергию.

Комбинируя эти команды, мы можем решить любые задачи.

Важно еще отметить, что существует продукт LAUNCHXL-CC13-90, который отличается тем, что имеет усилитель сигнала. Сам усилитель управляется с помощью GPIO. Наш драйвер будет тоже поддерживать эти устройства.

Итак, рассмотрим настройку радио модуля. Он поддерживает следующие диапазоны частот:

- 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

А также следующие скорости передачи данных:

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

Для инициализации следует вызвать функцию **radio_init**. Она получает следующие параметры: скорость передачи данных, диапазон частот, использовать усилитель сигнала или нет, отклонение частоты и пропускная способность. Последние 2 параметра можно задать deviation=0xFFFF и rx_bandwidth=0xFF. В этом случае они будут вычисляться автоматически согласно правилу пропускной способности Карлсона. Это правило определяет отношение между отклонением и пропускной способностью.

```
    /* 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
```

В начале функции radio_init мы проверяем параметры и в случае использования усилителя, инициализируем нужные GPIO. Далее в случае использования LED-а для индикации активности радио модуля, мы привязываем его к выводу радио модуля.

```
#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
```

Далее инициализируем очередь входящих пакетов.

```
    /* 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);
    }
```

Инициализируем параметры радио.

```
/* 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;
    }
```

Для использования радио используется функция **radio_open**. Она используется один раз при запуске программы. По сути, она запускает настройку радио модуля согласно параметрам, которые мы задали в функции radio_init.

Функция **radio_standby** переводит радио модуль в режим энергосбережения. В этом режиме прием и передача сообщений невозможна.

Функция **radio_close** закрывает драйвер и освобождает ресурсы. После ее вызова следует вызвать функцию radio_open если нам нужно использовать модуль снова.

Фукции **radio_register_message_received_callback, radio_register_message_sent_callback, radio_register_command_completed_callback, radio_register_rssi_measured_callback** используются для отслеживания событий драйвера.

Функция **radio_set_channel** используется для задания канала в диапазоне частот. По сути, она задает частоту в выбранном диапазоне.

Функция **radio_set_tx_power** задает уровень мощности передачи сигнала. Также есть функции **radio_decrease_tx_power** и **radio_increase_tx_power**, которые изменяют уровень мощности на одну позицию.

Функция **radio_set_sync_word** определяет Sync word.

Функция **radio_get_time** используется для получения времени радио модуля.

Рассмотрим функции приема и передачи пакетов. Прием пакетов реализован в функции **radio_enable_receive_internal**.

Она имеет следующие параметры:

start_time – время запуска приема (время по часам радио модуля).

timeout_us – таймаут в микросекундах.

receive_on_timeout – продолжить прием в случае достижения таймаута и частичного получении пакета.

use_start_time – использовать время начала или запустить немедленно.

continue_on_receive – продолжить прием в случае получении пакета и недостижении таймаута.

Для начала она останавливает любую активность (прием или передача), в случае если они не окончены.

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

Далее задаем параметры приема.

```
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;
```

И запускается прием.

```
/* 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;
}
```

Отправка пакетов реализован в функции **radio_send_internal**.

Она имеет следующие параметры:

data – Указатель на массив передаваемых данных.

size – Размер передаваемых данных.

timestamp – Время начала передачи (время по часам радио модуля).

lbt_rssi – Уровень шума для определения передачи в случае LBT.

lbt_timeout_us – Таймаут LBT в микросекундах. Время проверки шума. 0 – не использовать LBT.

use_start_time – использовать время начала или запустить немедленно.

Для проверки драйвера создадим два теста. Первый будет посылать пакеты данных с частотой пакет в 5 секунд. При окончании отправки выводится сообщение на терминал с количеством отправленных сообщений. Второй тест будет принимать пакеты и сообщать через терминал о количестве принятых пакетов. Так же при активности радио будет загораться красный LED.

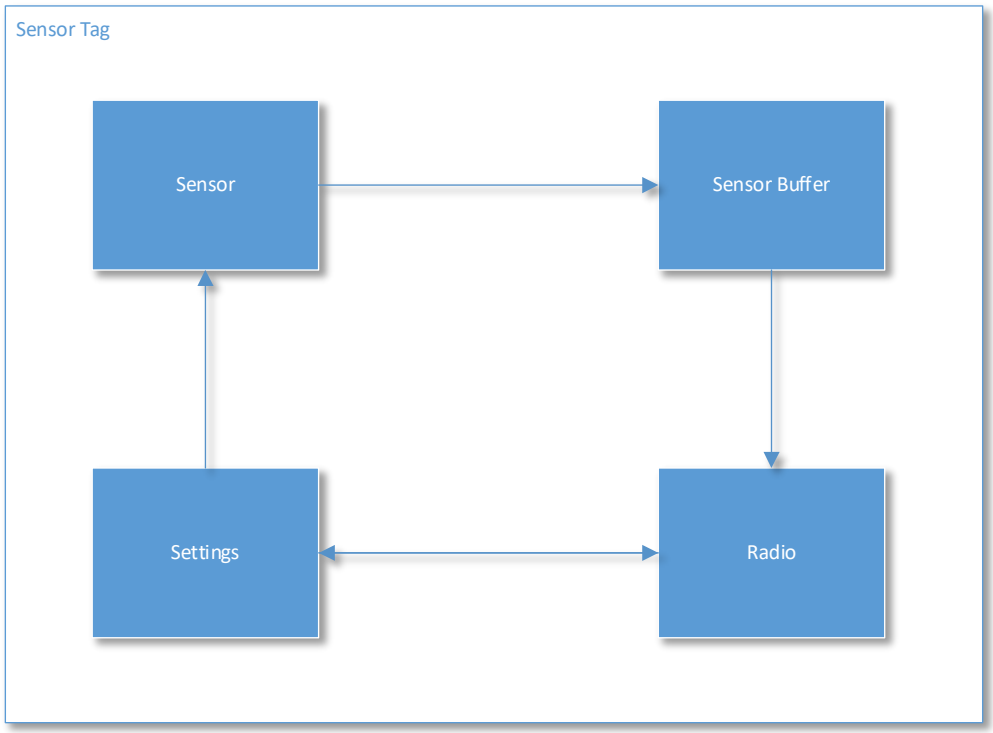
IoT проект

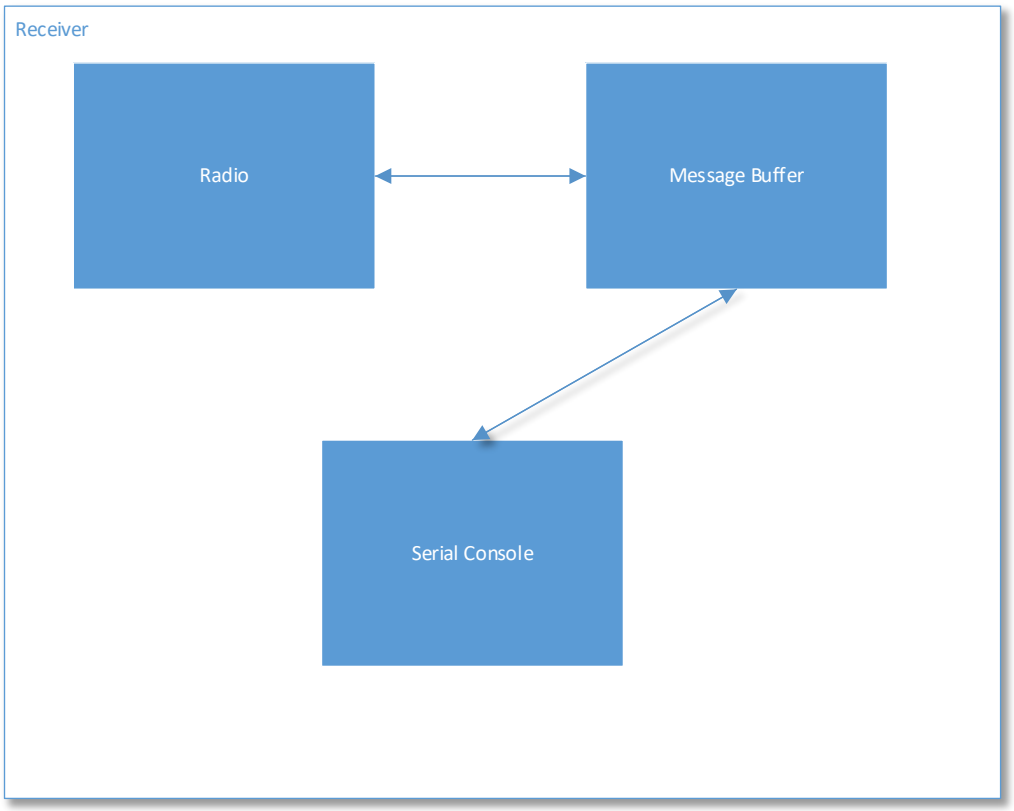
Чтоб почувствовать, как работает IoT, мы создадим проект. Идея проекта заключается в том, что у нас есть беспроводные датчики/устройства, которые работают от батарейки и посылают информацию в определенный период времени. Так же имеется базовое/центральное устройство, которое подключено к компьютеру, принимающее данные от датчиков и выводящее полученные данные через терминал. Так же есть возможность изменить отчетный период датчика, другими словами, как часто датчик будет посылать данные. Для простоты датчик будет посылать напряжение батарейки (или напряжение питания в случаи если питается не от батареи). Так же будет возможность послать данные датчика, при нажатии кнопки на нем.

Дизайн проекта

Многим молодым программистам не терпится сразу бросится программировать. Но не стоит спешить. Важно сперва продумать структуру проекта, продумать из каких модулей он будет состоять и как эти модули взаимодействуют друг с другом. Разделяя большой проект на маленькие части, даст нам возможность лучше продумать и реализовать каждую часть. Это поможет избежать тех моментов, когда придется переделывать все, и вся проделанная работа пойдет «коту под хвост».

Итак, у нас есть два вида устройств, то есть два приложения. Первое это принимающее устройство, назовем его Gateway, и второе, непосредственно датчик. Gateway должен находится в постоянном приеме, чтоб не пропустить данные с различных датчиков. Следовательно он не может заходить в режим энергосбережения и питаться от батарейки. С другой стороны датчик большую часть времени находится в «спящем» режиме и просыпается только тогда, когда ему нужно послать данные. Для простоты понимания изобразим модули и связи между ними на следующей блок-диаграмме.





Все модули должны работать независимо от других. Это даст нам возможность проверять каждый модуль отдельно. Связывать модули будет уже конечное приложение. Каждый модуль должен предлагать удобный интерфейс для работы с ним.

Настройки приложения

В каждом приложении имеются параметры, которые изменяются по мере использования. Например, частота с которой проводятся измерения датчика, в нашем случае это - отчетный период датчика. При перезапуске устройства этот параметр должен не изменяться, то есть он должен храниться в постоянной памяти. За этот функционал будет отвечать модуль настроек (Settings). Параметры будут храниться как регистры. Это даст нам возможность просто добавить необходимый параметр в будущем, в случае надобности.

Буфер датчика

Буфер датчика (Sensor buffer) будет хранить в себе данные, полученные от датчика, и по мере возможности будут отсылаться на принимающее устройство. Буфер имеет структуру очереди, что значит, что элемент, попадающий первым в очередь, покидает ее тоже первым.

Буфер сообщений

Буфер сообщений (Message buffer) будет хранить в себе данные, полученные от датчиков через радио, и по мере возможности будут отсылаться на консоль. Буфер имеет структуру очереди, что значит, что элемент, попадающий первым в очередь, покидает ее тоже первым. Так же этот буфер имеет 2 очереди: принятые сообщения и сообщения на отсылку.

Радио

Этот модуль (Radio) отвечает за передачу данных по радио. Он содержит в себе автомат состояний, согласно которому будут передаваться/приниматься данные. Поскольку в нашем проекте два вида устройств, и каждое работает не так как другое, нам нужно будет изменить работу автомата состояний в зависимости от вида устройства. Так же для устройства «Датчик» во время простоя должен поддерживаться режим энергосбережения.

Консоль

Этот модуль (Serial Console) отвечает за передачу данных через серийный порт между приемным устройством и компьютером пользователя. Он так же, как и модуль радио содержит в себе автомат состояний.

Датчик

Этот модуль (Sensor) отвечает за получение данных от датчика. Он содержит в себе автомат состояний, который по мере надобности будет запускать процесс чтения датчика.

Реализация проекта

Итак, приступим к реализации проекта. В процессе мы будем реализовывать модули и проверять их с помощью тестов. По мере реализации всем модулей, создадим конечные приложения.

Модуль настроек (Settings)

Данные настроек хранятся в ПЗУ, в нашем случае это FLASH. В связи с тем, что программа так же располагается во FLASH памяти, настройки мы будем хранить в конечных адресах памяти. CC1310 LaunchPad™ использует микропроцессор с FLASH памятью 128 кБ, это 32 страницы. Последнюю страницу мы не будем использовать, так как она содержит в себе CCFG, и при стирании страницы CCFG обнулится. Конечно, это можно предотвратить, добавив код, но в этом курсе нам будет проще использовать страницу 31 взамен.

Итак, для хранения настроек создадим следующую структуру:

```
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;
};
```

Эта структура в заголовке имеет так называемый якорь (или ориентир), по которому мы сможем найти ее во FLASH памяти. В конце структуры имеются два байта контрольной суммы, она нам нужна для верификации целостности настроек. Обратите внимание на макрос ATTRIBUTE_PACKED_ALIGNED(1). Он находится в файле ti-lib.h и, по сути, содержит атрибут компилятора, который указывает на то, что структура использует выравнивание в один байт, а не в 4 (по умолчанию). Это значит, что в случае выравнивания по умолчанию, размер структуры будет кратным 4 даже если ее размер один байт. Выравнивание по умолчанию в 4 байта сделано не просто так, причина этого то, что микроконтроллер 32-битный и ему проще и быстрее выполнять операции с переменными размером 32 бита (4 байта).

Для вычисления контрольной суммы будем использовать алгоритм CRC-16/XMODEM. Конечно, можно и использовать любой другой алгоритм, но мы остановимся на нем. Для подсчета контрольной суммы создадим функцию calculate_crc16_xmodem в файле common_utilities.

Модуль имеет 3 функции:

settings_load – загрузка настроек из FLASH памяти в переменную settings.

settings_save – сохранение настроек из переменной settings во FLASH память.

settings_set_default – применение настроек по умолчанию. Эта функция не сохраняет настройки во FLASH память.

Рассмотрим функцию settings_load**.** Она использует функцию settings_find_pointer, которая ищет адрес якоря на странице настроек. В случае если адрес не найден (за пределами страницы), то функция settings_load вернет false, в противном случае она загрузит настройки из FLASH памяти в структуру, подсчитает контрольную сумму и сравнит ее с суммой, хранимой в самой структуре. Результат сравнения будет результатом функции.

Следующая функция - settings_save. Она так же использует функцию settings_find_pointer. В случае если она вернет адрес в пределах страницы, то мы обнуляем все биты структуры хранящейся во FLASH памяти, продвигаем указатель на следующую позицию и записываем туда структуру настроек из переменной. В противном случае мы очищаем страницу и пишем структуру в начало страницы. Обратите внимание на то, что если указатель, который вернет функция settings_find_pointer равен последнему возможному положению структуры настроек, то и в этом случае мы очищаем страницу, потому что если мы продвинем указатель на следующую позицию, то мы выйдем за пределы страницы. Так же, обратите внимание на то, что перед записью во FLASH память, мы пересчитываем контрольную сумму, чтоб она сохранилась обновленной.

Функция settings_set_default ставит настройки по умолчанию в переменную settings и пересчитывает контрольную сумму.

Чтоб проверить модуль, создадим тест settings_test. Нам важно проверить поведение модуля в случае достижения конца страницы настроек. Для этого мы будем изменять настройки, сохранять, и считывать их обратно. Потом сверяем сохранённые настройки с теми, что сохранили и выводим результат сравнения на консоль. Эту процедуру будем повторять 300 раз, так как структура имеет размер 14 байтов, а за 300 итерации мы заполним всю страницу (общее количество байтов = 4200). Обратите внимание на то, что за 300 итерации страницу мы стерли всего один раз, тем самым продлив жизнь FLASH памяти.

Буфер датчика (Sensor buffer)

Данные, полученные с «датчика», мы будем хранить в ОЗУ используя циклический буфер. Другими словами, будем использовать очередь, которая хранит данные, полученные с датчика. По сути, в очередь мы будем писать структуры sensor_buffer_record_s. Это структура имеет следующий вид:

```
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];
};
```

Она имеет заголовок содержащий код сенсора и размер записи. После заголовка хранятся непосредственно данные. Максимальный размер данных 12 байт. Размер буфера датчика будет 300 байт. Эти параметры можно будет легко изменить в последствии.

Для самого буфера будем использовать вспомогательную структуру:

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

Она состоит непосредственно из объекта циклического буфера и количества хранящихся записей.

Также модуль будет иметь следующие функции:

sensor_buffer_initialize – Инициализация буфера.

sensor_buffer_get_records_count – Получение количества записей в буфере.

sensor_buffer_pop_record – Получение записи из буфера.

sensor_buffer_push_record – Добавление записи в буфер.

Рассмотрим каждую из них. При инициализации буфера мы инициализируем циклический буфер и обнуляем счетчики.

sensor_buffer_get_records_count попросту возвращает количество из счетчика.

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;
}
```

сперва проверяет есть ли достаточно места чтобы добавить новую запись. В случае если места недостаточно, достает записи пока не будет достаточно свободного места. Далее она копирует байты структуры с буфер, увеличивает число записей и выходит.

sensor_buffer_pop_record –

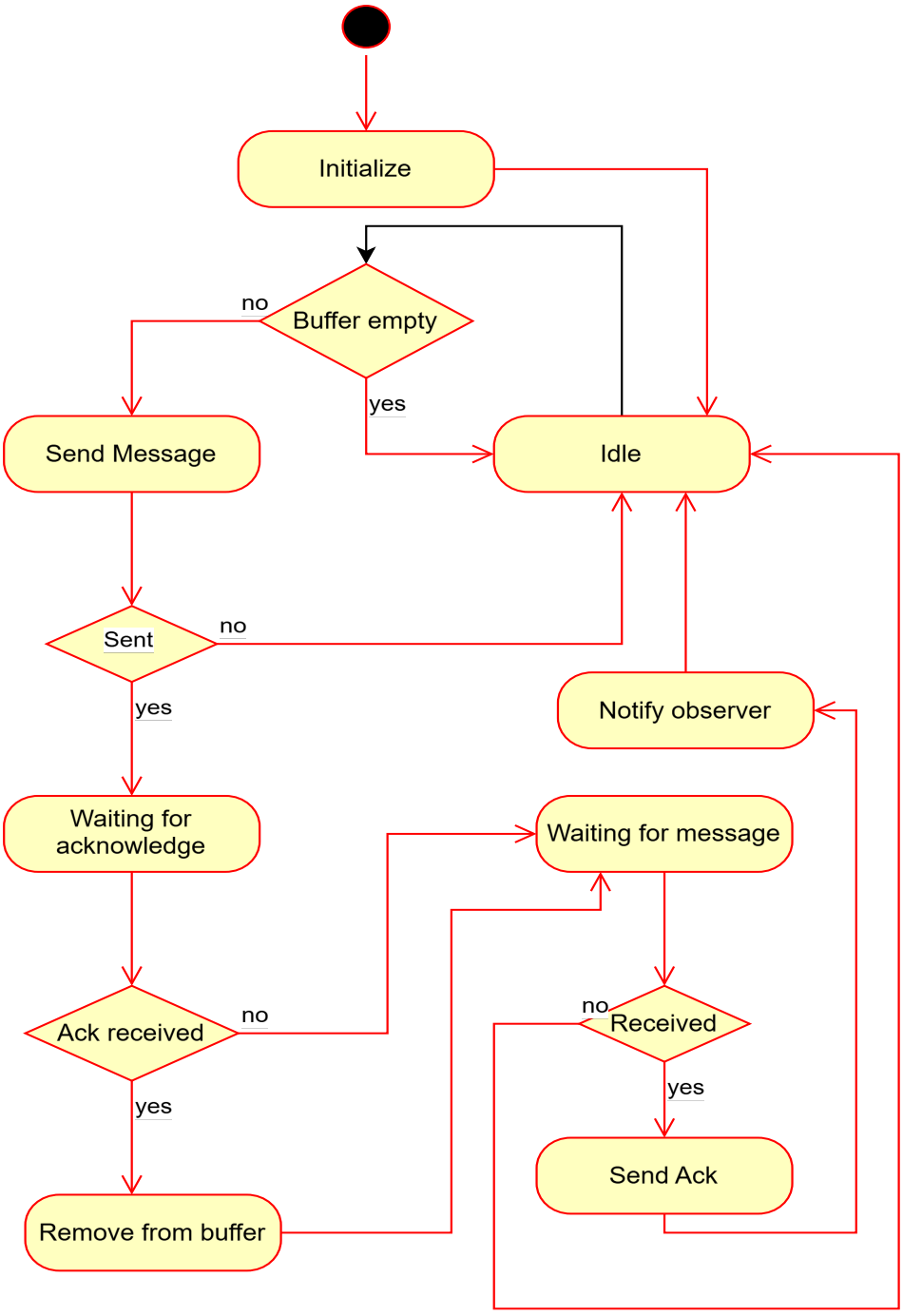
```
bool sensor_buffer_pop_record(sensor_buffer_record_union_p record)
{
    size_t i;
    if (record == NULL || sensor_buffer.buf_count == 0)
    {
        return false;
    }
    memset(record->array, 0, SIZE_OF_ARRAY(record->array));
    for (i = 0; i < SENSOR_RECORD_HEADER_SIZE; i++)
    {
     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

Сперва проверяет есть ли записи в буфере, в случае что есть то сперва копирует заголовок, а потом данные, в зависимости от размера данных, записанных в заголовке.

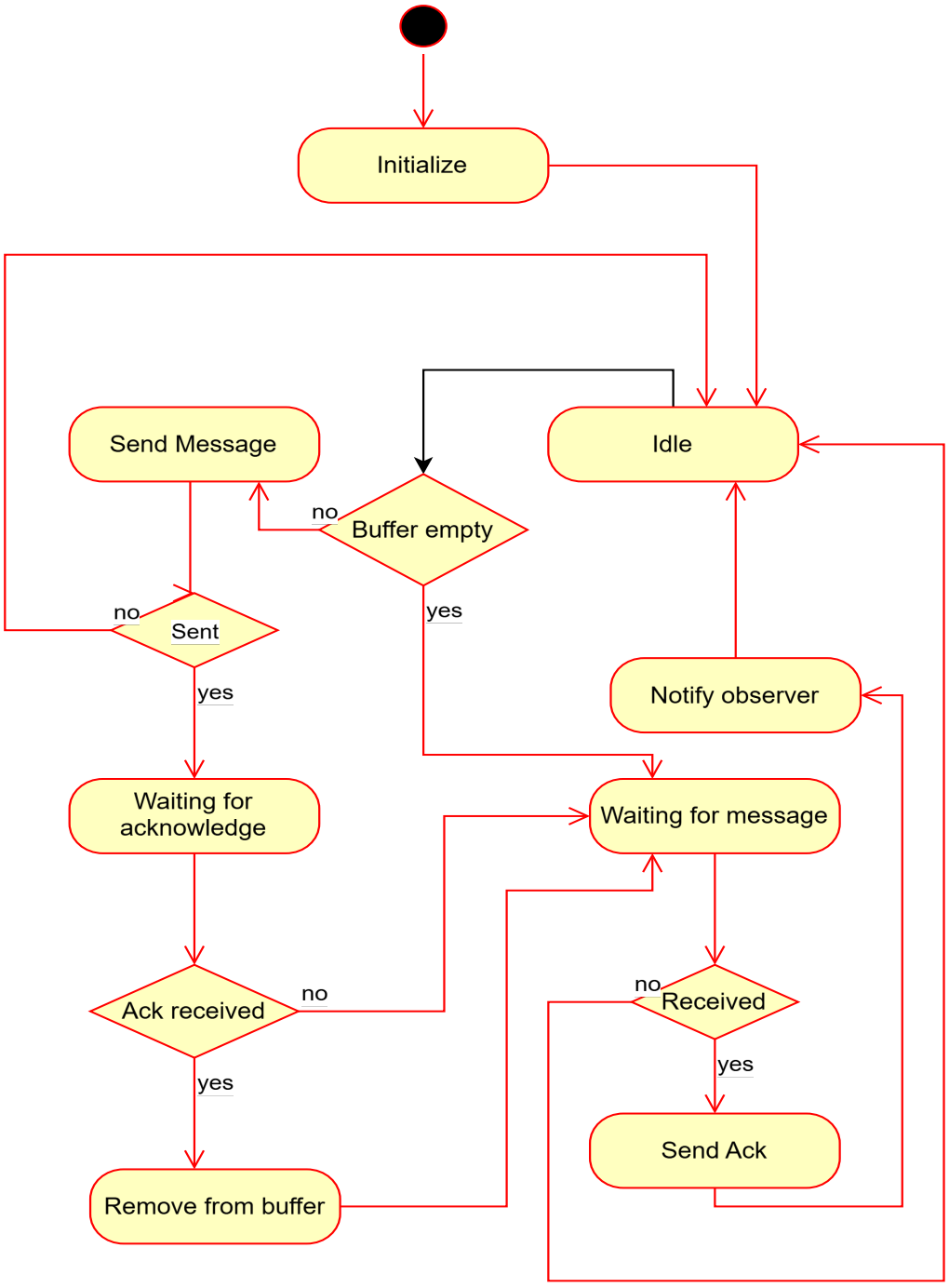
Чтоб проверить модуль, создадим тест sensor_buffer_test. Он будет сохранять данные в буфер, и считывать их обратно. Потом сверяем сохранённые настройки с теми, что сохранили и выводим результат сравнения на консоль. Эту процедуру будем повторять 300 раз.

Радио протокол (Radio Protocol)

Как нам известно из проектирования проекта, у нас имеются два вида устройств: приемное устройство и передатчик.
Рассмотрим автомат состояний передатчика. У нас имеется буфер исходящих сообщений, в который датчик будет добавлять сообщения. В случае, если он пуст, автомат перейдет в состояние ожидания (IDLE). В этом состоянии мы можем перевести устройство в спящий режим.
При появлении сообщения в буфере, автомат перейдет в режим отправки. В случае успешного отправления, автомат перейдет, а режим приема подтверждающего сообщения (Acknowledge) от получателя, в противном случае – обратно в режим ожидания. При получении Acknowledge-а, автомат удалит сообщение из буфера. Далее автомат перейдет в режим ожидания сообщения. В случае получения сообщения, автомат перейдет в режим отправки Acknowledge, а затем оповестит о получении сообщения и обратно в режим ожидания.



Принимающее устройство имеет похожий автомат состояний, с разницей в том, что он не переходит в спящий режим, а переходит вместо этого в режим ожидания сообщения.



Теперь рассмотрим структуру сообщений. Каждое сообщение имеет заголовок, который идентичен для всех видов сообщений:

```
/**
 * @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];
};
```

Заголовок имеет следующие поля:

- Размер сообщения в байтах
- Тип сообщения
- ID устройства отправителя
- ID устройства получателя
- Номер сообщения (он нам нужен для Acknowledge)

Чтоб получить ID устройства мы будем использовать MAC адрес. Он состоит из 8 байтов, но мы возьмем нижние 4, этого будет достаточно. Чтоб получить MAC адрес воспользуемся функцией get_mac_address.

Обратите внимание что мы создали так же union. Это очень полезный «трюк» при работе с массивами данных, для создания сообщения мы будем использовать поле структуры (header в случае header-a), а при пересылке будем использовать поле массива (array).

Так же обратите внимание на атрибут ATTRIBUTE_PACKED_ALIGNED(1). Он используется для того, чтоб сказать компилятору выравнивать переменные в длину кратную 1 байту, а не 4 по умолчанию. Если мы используем выравнивание в 4 байта, то структуры будут занимать память размеров кратному 4 байтам, а в случае нашего заголовка у нас 11 байтов, что не является кратным 4.

Сообщение Acknowledge имеет следующий вид:

```
/**
 * @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];
};
```

Как можно видеть, сообщение состоит из заголовка и номера сообщения, которое оно подтверждает.

Сообщение Data имеет помимо заголовка массив данных. Максимальный размер массива задан с помощью 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];
};
```

Сообщение команды имеет следующий вид:

```
/**
 * @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[7];
};

/**
 * @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[20];
};
```

Оно имеет следующие поля:

- Направление – Запрос или ответ на запрос.
- Команда – Читать или писать.
- Параметры – параметры команды. Мы будем использовать «таблицу поиска» или, другими словами, регистры. Каждый регистр ответственен за какой-то параметр настроек. Параметры могут быть номер регистра и его значение. Так же при ответе на запрос мы будем возвращать код результата (0 – ошибка, 1 - OK).

Чтоб различать два вида автомата состояний в коде будем использовать макрос ROOT_NODE.

Рассмотрим основные состояния автомата состояний радио протокола. Как можно видеть из файла radio_protocol.c, переключение состояний происходит в основной функции radio_protocol_process, либо в обработчиках событий радио драйвера.

Состояние инициализации – это состояние запускается при запуске программы. Оно настраивает радио модуль на нужную частоту и скорость согласно макросам, заданным в начале файла или посредствам файла команд. Также подключаем обработчики событий радио драйвера и переводим автомат в состояние ожидания.

Состояние ожидания – в этом состоянии мы определяем какая следующая команда. Она зависит от того есть ли в буфере сообщения на отсылку (в данном примере для простоты, мы будем использовать буфер в одно сообщение), если буфер не пуст – то переводим автомат в состояние отправки, в противном случае – в состояние приема (для приемника) или в состояние ожидания (для датчика).

Состояние приема – в этом состоянии мы запускаем прием сообщений. Важно отметить, что передатчик имеет таймаут в одно сообщение. Если за это время сообщение не было получено, то устройство перейдет в спящий режим.

Конечно, данный автомат состояний не идеален, так как в случае неполучения Acknowledge-а, он может посылать сообщения без остановки, но для образовательных целей этого нам достаточно.

Состояние отправки - в этом состоянии мы запускаем отправку сообщения в случае, если оно имеется. В противном случае автомат перейдет в состояние ожидания, а в случае приемника – в состояние приема. Также стоит отметить, что при посылке сообщения мы будем использовать механизм LBT (Listen

before talk), чтоб избежать коллизии между другими устройствами, а отправка Acknowledge-a – без, потому что ответ должен быть мгновенным.

Состояние обработки сообщения – в этом состоянии автомат обрабатывает принятое сообщение и уведомляет слушателя (главное приложение) о получении сообщения путем вызова обработчика события.

Так же у нас имеются вспомогательные функции:

- radio_protocol_is_active – показывает занят ли автомат состояний, или можно переходить в спящий режим.
- radio_protocol_has_outgoing_message – имеется ли сообщение на отправку.
- radio_protocol_append_data_message – добавляет сообщение типа Data на отправку.
- radio_protocol_append_command_message - добавляет сообщение типа Command на отправку.
- А также функции регистрации обработчиков событий.

Монитор питания (Battery Monitor)

Из описания проекта, устройство с датчиком должно посылать напряжение батарейки. Чтоб получить напряжение батарейки воспользуемся встроенным модулем микропроцессора (AON_BATMON), который позволяет получить входное напряжение микроконтроллера и его температуру. Модуль реализован в файле bat_monitor_sensor. В связи с тем, что замер напряжения и температуры происходит не мгновенно, модуль будет использовать автомат состояний, чтоб не мешать работе других модулей блокировкой во время измерения. Автомат имеет всего три состояния:

- BAT_MONITOR_SENSOR_STATE_SAMPLE – Начало измерения.
- BAT_MONITOR_SENSOR_STATE_SAMPLE_STARTED – Измерение в процессе.
- BAT_MONITOR_SENSOR_STATE_NONE – Ожидание.

При окончании измерения модуль вызывает обработчик события и передает в него результат измерения (время измерения с момента запуска, напряжение в милливольтах и температуру в градусах).

Для сбережения энергии модуль останавливает измерение AON_BATMON-а.

Кнопка (Push Button)

Этот модуль отвечает за обработку нажатия кнопки. В проекте мы будем использовать кнопку BTN-2 комплекта разработчика. Модуль реализован в

файле push_button. При нажатии на кнопку и освобождении вызываются обработчики событий. Важно отметить, что срабатывание кнопки работает даже в «спящем» режиме, поэтому если мы хотим послать сообщение, нам следует вывести микроконтроллер из спящего режима с помощью макроса SLEEP_BREAK.

Приложение устройства с датчиком (Tag)

Для создания приложения устройства с датчиком у нас уже есть все необходимые модули. Рассмотрим реализацию приложения. Его код находится в файле tag.c. Инициализация приложения начинается с загрузки настроек.

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

В случае неудачной загрузки, мы применяем настройки по умолчанию.

Далее, чтоб уведомить пользователя о запуске программы, поморгаем два раза LED-ами. Эта практика часто используется во встроенных системах.

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

Далее инициализируем радио модуль, буфер датчика и датчик, и привязываем обработчики событий.

```
    /* 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);
```

В главном цикле мы вызываем функции обработки в нужных модулях и в случаи простоя – посылаем устройство «спать». Время сна зависит от того, есть ли сообщения на отправку. В случае, что таковы имеются – время будет равно 1 секунде.

```
    /* 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);
        }
    }
```

При обработке буфера датчика, мы проверяем, есть ли в нем сообщения, и в случае того, что он не пустой, достаем сообщение и передаем его модулю радио на отправку.

```
    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);
        }
    }
```

Похожая ситуация у нас в обработчике входящих команд. Мы проверяем если получен запрос на чтение или запись регистра, то выполняем команду, генерируем результат и передаем его на модуль радио. Стоит отметить, что при записи единицы в регистр 0 – произойдет перезапуск устройства. Так мы сможем удаленно перезапускать устройство.

```
    /* 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);
        }
    }
```

При нажатии на кнопку сработает обработчик события кнопки. В нем мы запустим новый замер напряжения и поднимем флаг. В случае если нажатие произошло во время «сна», мы выводим микроконтроллер из спящего режима.

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

И при завершении измерения, создаем запись в буфере датчика. В случае поднятого флага, выводим микроконтроллер из спящего режима и опускаем флаг.

```
    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();
    }
```

Модуль консоли (Console Protocol)

Приемное устройство выводит принятые данные от передатчиков через консоль. Так же для конфигурирования передатчиков используются команды, полученные через консоль. Модуль консоли отвечает за передачу данных через консоль. Как и говорилось раньше, изменять параметры датчиков мы будем с помощью команд регистров, а именно чтение и запись регистров.

Команда чтения имеет следующий формат:

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

Пример команды:

GETREG:329563755,1<CR>

Данная команда читает регистр 1 датчика с номером 329563755. Регистр 1 – это частота чтения датчика в секундах.

Команда чтения имеет следующий формат:

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

Пример команды:

SETREG:329563755,1,60<CR>

Данная команда пишет значение 60 в регистр 1 датчика с номером 329563755.

Так же есть возможность перезапустить устройство удаленно. Для этого нужно записать значение 1 в регистр 0 устройства.

Рассмотрим реализацию модуля. Он находится в файле console_protocol.c. Для инициализации модуля используется функция console_protocol_initialize. Она настраивает драйвер UART. Настройки серийного порта по умолчанию:

Baudrate: 19200

Parity: None

Data Bits: 8

Stop Bits: 1

А также при инициализации привязываем обработчик событий полученных сообщений. В нем будет разбираться полученное сообщение, и в случае что это команда чтение или запись регистра, будет вызваться обработчик события, чтоб уведомить главное приложение.

Так же есть две функции для вывода результатов в текстовом формате.

console_protocol_send_command_message – выводит результат чтения/записи регистров.

console_protocol_send_sensor_data_message - выводит результат чтения датчика.

Приложение устройства приёма (Gateway)

Для создания приложения устройства, которое будет принимать данные датчиков, рассмотрим файл gateway.c. Инициализация похожа на инициализацию приложения Tag с одной лишь разницей, что нет инициализации датчика, а вместо этого инициализация модуля консоли.

```
/* 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);
```

В главном цикле мы вызываем функции обработки в нужных модулях. Обратите внимание, что в этом приложении устройство не заходит в «спящий» режим.

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

        /* Process radio protocol */
        radio_protocol_process();

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

Обработчики событий модуля радио вызывают функции модуля консоли, чтоб уведомить пользователя об полученных сообщениях. В случае, когда сообщение – результат команды, проверяется если это команда ответа (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);
    }
}
```

С другой стороны, обработчик события модуля консоли, пересылает команду на модуль радио, который в свою очередь перешлет её на нужное устройство.

Заключение

Вот мы и создали наш первый IoT проект. Конечно, он далеко от идеального, но всегда стоит начинать с чего-то простого и постепенно усложнять задачи. Надеюсь, этот проект послужит вам хорошим началом работы в столь увлекательном мире IoT. Буду рад получить ваше мнение о проделанной мной работе, ведь всегда полезно получать отзыв о проделанной работе, даже если этот отзыв не совсем приятный. Связаться со мной можно по почте: kulipator@gmail.com.