СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ИНЖЕНЕРНОЙ ФИЗИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра
«Радиоэлектронная техника
информационных систем»

Утверждаю:
Зав. кафедры
Гребенников А. В.

Лабораторный практикум
«Цифровые устройства и микропроцессоры»

СОДЕРЖАНИЕ

Введение

Настоящее методическое пособие направлено на укрепление знаний в области цифровой схемотехники, архитектуры процессорных систем, освоение языка программирования Ассемблера и программной среды Keil MDK.

Лабораторная работа №1. Изучение языка ассемблера, ядра Cortex-M3 и среды разработки Keil MDK.

Введение

Данная лабораторная работа предназначена для освоения инструкций ассемблера микропроцессора Cortex-M3 и программной среды Keil MDK.

1. Программный пакет разработки для ARM- микроконтроллеров Keil MDK

Интегрированная среда разработки (Integrated Development Environment) —µVision IDE — фирмы Keil сочетает в себе следующие возможности:

– управление проектами;

– создание отдельных программ;

– редактирование текста программы;

– отладка программ, позволяет непосредственно вызывать симулятор или внутрисхемный эмулятор.

Редактор и отладчик объединены в одно приложение, что упрощает процесс разработки проекта. µVision проста в использовании содержит богатый набор опций:

– **Device Database.** Интеллектуальная база данных с детальной информацией обо всех контроллерах, поддерживаемых инструментальными средствами Keil. База данных автоматически конфигурирует ассемблер, компилятор C/C++ и компоновщик для выбранного микроконтроллера, генерирует файлы описания регистров, конфигурирует симулятор CPU и периферии, корректирует код инициализации и программные алгоритмы. Device Database содержит подробные инструкции по конфигурированию, ссылки на другие источники информации включает более чем 200 ARM-микроконтроллеров (полный список этих устройств можно найти на сайте www.keil.com/dd).

– **Project Manager.** Менеджер проекта, дает методику создания проекта из исходных файлов, различных опций разработки и директорий. Программный проект состоит из большого числа файлов, которые обрабатываются индивидуально. Например, часть файлов подлежит компиляции, а другие следует ассемблировать. При этом достигается простая интеграция различных исходных файлов в проект.

– **Building Projects.** Менеджер проекта, позволяет создавать в одном проекте отдельные файлы для симуляции, отладки с помощью программы-эмулятора и программирования EEPROM. Ассемблер и компилятор автоматически генерируют зависимости между файлами и добавляют их в проект. При глобальной оптимизации µVision неоднократно компилирует исходный файл для достижения оптимального использования регистров. Все параметры проекта сохраняются в специальном файле, то есть компиляция и линковка проекта происходят по нажатию одной клавиши.

– **Отладчик симулятор µVision Debugger.** Полнофункциональный отладчик,

который позволяет вести отладку программ, написанных на С и ассемблере или в смешанном формате, а также сделать выбор между симулятором, монитором, JTAG-отладчиком и внутрисхемным эмулятором.

– **µVision Editor**. Интегрированный редактор облегчает подготовку исходного текста за счет многооконности, выделения синтаксиса цветом и исправления ошибок в режиме диалога. Редактор настраивается в соответствии со вкусами пользователя. Интерактивная система исправления ошибок позволяет отслеживать ошибки и предупреждения, которые появляются в отдельном окне во время отладки программы. Существует возможность исправления файлов проекта, пока µVision продолжает проверку в фоновом режиме. Номера строк, содержащих ошибку или предупреждение, автоматически обновляются при изменении исходного файла.

– **µVision Utilities**. Мощные интегрированные утилиты, облегчающие создание проекта. Source Browser — база данных программных символов для быстрой навигации по исходному файлу, Find in Files —полный поиск во всех файлах, PC-Lint — анализ синтаксиса исходного кода, Flashtool — утилиты загрузки флэш-памяти и многие другие.

– **On-line help**— встроенная система помощи, содержит как краткую информацию об использовании программного обеспечения, так и полный перечень руководств пользователя On-line Manuals.

1.1. Установка Keil MDK

1.1.1. Зайдите на сайт http://www2.keil.com/mdk5 и скачйте последнюю версию программы (рисунок 1).

Рис 1. WEB страница для загрузки установщика MDK Keil.

1.1.2. Введите свои личные данные для скачивания (рисунок 2).

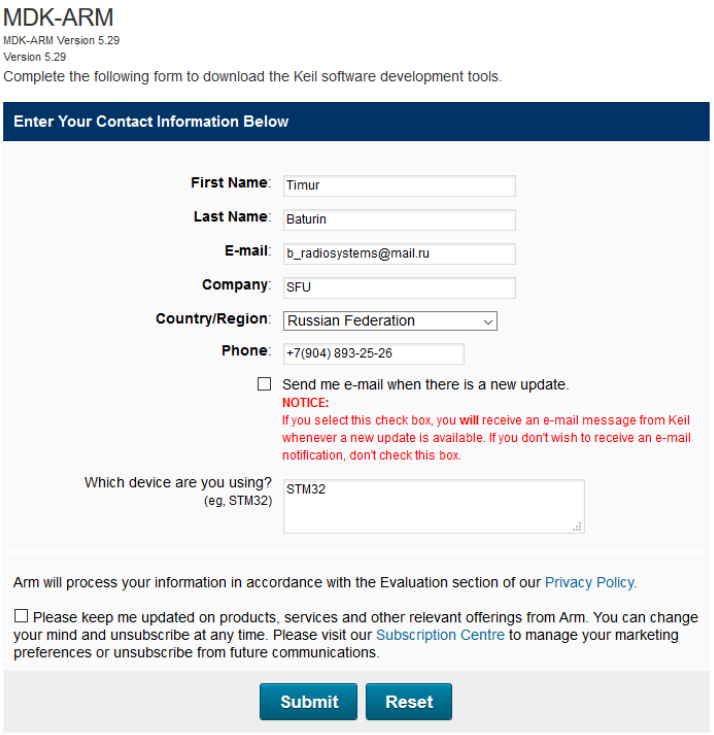

Рис 2. Пример заполнения формы для скачивания установщика.

1.1.3. Нажмите на кнопку, показанную на рисунке 3.

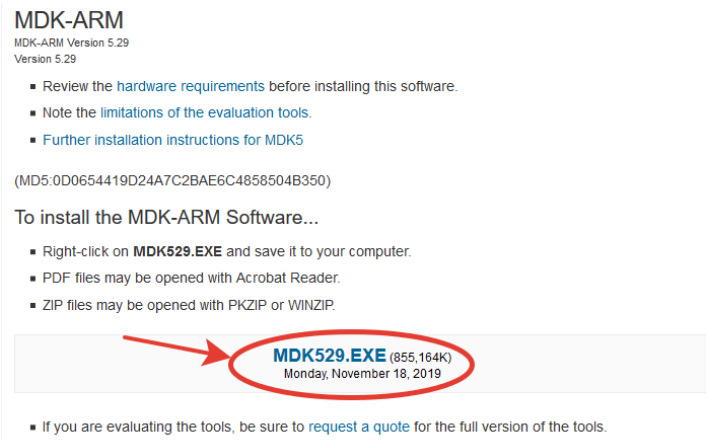

Рис 3. Кнопка для скачивания установщика.

1.1.4. После завершения загрузки запустите установщик и установите программу MDK Keil.

1.1.5. После установки запустится установщик пакетов «Pack installer». В правом окне «Device» выберите микроконтроллер под которй вы собираетесь писать программу. Для данного курса выберите микроконтроллер «STMicroelectronics/ STM32F103/ STM32F103C8» и нажмите кнопку «Packs/ Check For Updates» (рисунок 4). Дождитесь обновления.

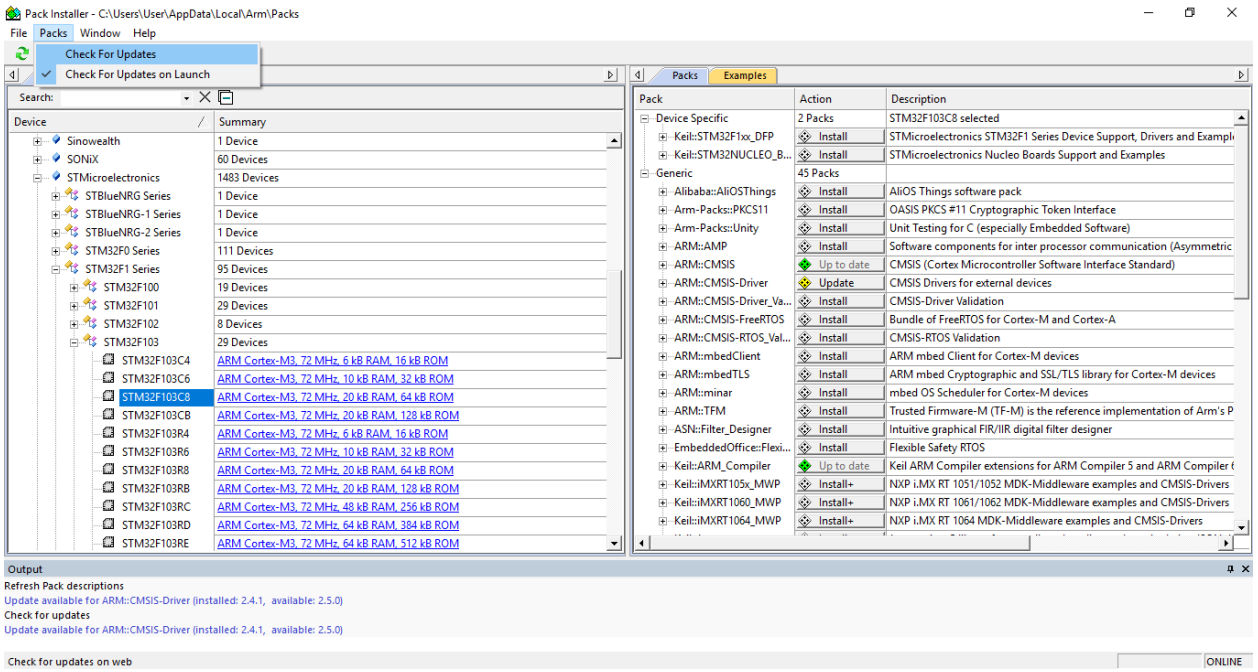

Рис 4. Установщик пакетов.

1.1.6. Нажмите копку «Install» в окне «Packs» показанную на рисунке 5.

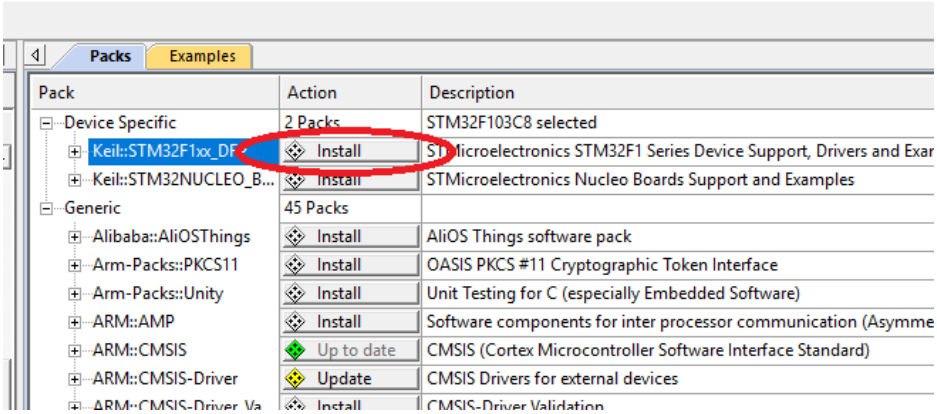

Рис 5. Установщика библиотеки для STM32F103.

Дождитесь завершения установки и закройте «Pack Installer»

1.1.7. Запустите Keil uVision5.

1.1.8. Зайдите во вкладку «Edit/Configuration». Поменяйте кодировку на UTF- и установите галочки в соответствии с рисунком 6.

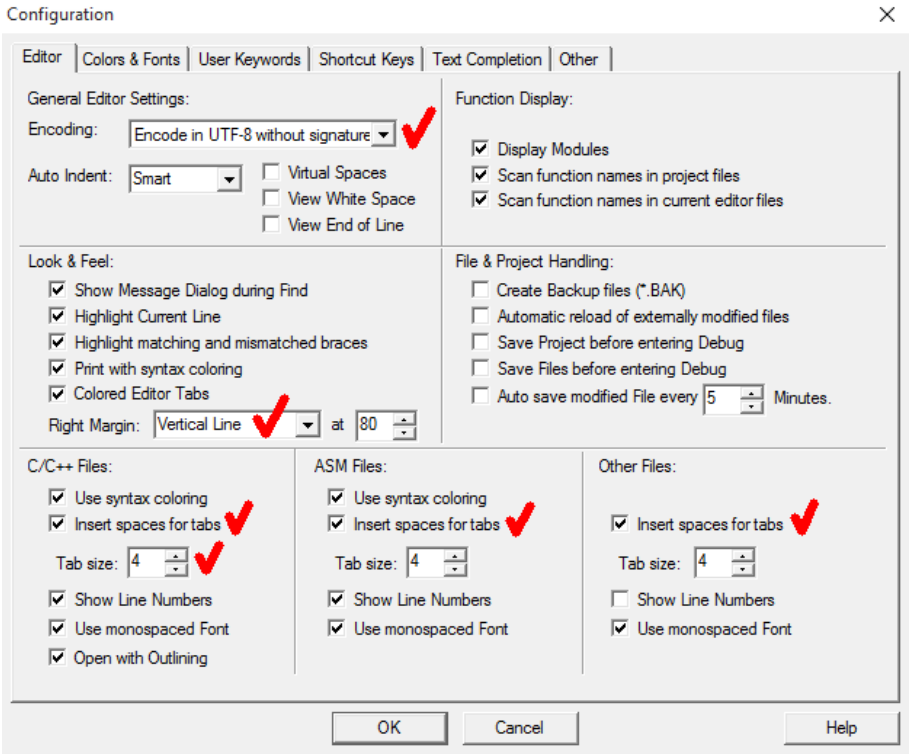

Рис 6. Настройка текстового редактора.

1.1.9. Установка завершена.

1.2. Отладчик симулятор uVision Debugger

µVision Debugger — это полнофункциональный отладчик, который позволяет вести отладку программ, написанных на C и ассемблере или в смешанном формате, а также сделать выбор между симулятором, монитором, JTAG-отладчиком и внутрисхемным эмулятором.

Полная симуляция включает симуляцию системы команд и встроенной периферии (АЦП, ЦАП, таймеров, UART, CAN, I^2C, прерываний, внешних сигналов и I/O) плюс управление. Симуляция предоставляет дополнительные возможности, не достижимые при JTAG-отладке: точный временной расчет и детальный анализ исполнения программы при различных параметрах.

µVision Debugger предоставляет разработчику следующие возможности:

• Breakpoints — задание точек останова осуществляется через результат выражения или обращение к ячейке памяти/ переменной. Для редактирования и просмотра параметров контрольных точек служит окно Breakpoint. Точки останова могут остановить исполнение программы, запустить команду или сценарий отладчика.

• Memory & Register — просмотр областей памяти и состояний регистров в специальных окнах. Окно Serial I/O делает наглядной симуляцию последовательного ввода/вывода.

• Performance Analyzer — анализатор производительности, фиксирует время исполнения программных модулей. Задавая список модулей для анализа, пользователь получает диаграмму затрат времени на каждую часть программы.

• Code Coverage — анализатор эффективности кода локализует части программы, к

которым редко происходит обращение, что позволяет удалить ненужный код. Анализ эффективности кода осуществляется на уровне C и ассемблера. Подробная статистика: время исполнения, число обращений.

• Target Monitor — монитор, обеспечивает прямой интерфейс при отладке программ на плате и легко настраивается на любой микроконтроллер. Отладка ничем не отличается от режима симуляции. Требования к ресурсам микроконтроллера со стороны монитора минимальны.

• JTAG Interface — поддержка разнообразных опций отладки через интерфейс JTAG для связи с различными устройствами либо с помощью адаптера USB-JTAG.

• Real-Time Agent — это небольшой программный модуль на C (занимает в приложении пользователя около 1500 байт), который позволяет вести отладку «на лету» и не требует остановки системы. Коммуникация осуществляется через адаптер USB-JTAG ULINK2 или ULINK-ME. Отладка «на лету» дает возможность во время исполнения программы осуществить чтение и запись памяти, доступ к переменным, установку точек останова, Serial I/O (printf).

• Serial Wire Debug — двухпроводной интерфейс для процессоров, который заменяет стандартный интерфейс JTAG, предлагая дополнительно к его возможностям доступ к памяти в реальном времени без останова процессора и какого-либо резидентного кода. Serial Wire Viewer использует еще один дополнительный контакт и позволяет просматривать значения переменных и сообщения об отладке при работающем на полной скорости процессоре.

1.3. Компиляция

Средства компиляции RealView Compilation Tools состоят из компилятора C/C++, библиотеки MicroLib, ассемблера и компоновщика. RealView Compilation Tools для ARM транслирует исходные файлы на C в объектные файлы, которые содержат полную символьную информацию для отладки с помощью μVision Debugger или внутрисхемного эмулятора. Кроме объектных файлов компилятор генерирует файл листинга, который опционально может включать таблицу символов и перекрестные ссылки.

RealView Assembler (armasm) транслирует инструкции ARM и Thumb в объектные файлы, обрабатываемые Linker/Locater или Library Manager.

RealView Linker (armLink) (компоновщик) осуществляет генерацию и оптимизацию кода, объединяет объектные модули ARM, создает исполняемые программы, распознает ссылки и назначает абсолютные или фиксированные адреса для сегментов программы. На выходе линкера — абсолютные объектные модули для загрузки в μVision Debugger или Intel HEX файл для программирования устройств.

2. Создание проекта

2.1. Откройте «Keil uVision5».

2.2. Откройте вкладку «Project».

2.3. Нажмите «New uVision5 Project».

2.4. Выберите папку в которой будет расположен проект. Присвойте имя проекту и нажмите «Сохранить».

2.5. Выберите микроконтроллер «STM32F103C8», как показано на рисунке 7. И нажмите OK.

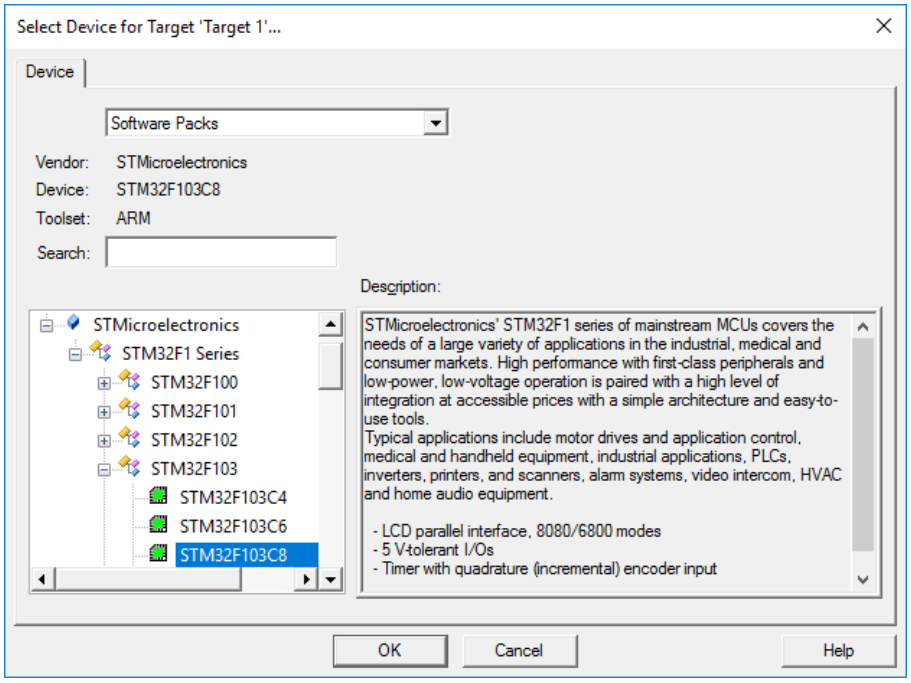

Рис 7. Выбор микроконтроллера при создании проекта.

2.6. В окне «Manage Run» ничего не выбираем и нажимаем кнопку OK. В окне «Project» наблюдаем созданный пустой проект.

2.7. Нажимаем правой кнопкой на папку «Source Group 1» и выбираем «Add New Item To Group …». Выбираем тип файла для ассемблера **.s** (Asm File). Вводим имя файла и нажимаем кнопку «Add».

2.8. Вставляем код из Листинга 1 в созданный файл с расширением **.s** :

Листинг 1. Код для запуска первой программы.

```
STACK_TOP EQU 0x20000100
    PRESERVE8
    THUMB

    AREA RESET, CODE, READONLY

    ; Таблица векторов прерываний
    DCD STACK_TOP      ; Указатель на вершину стека
    DCD Start          ; Вектор сброса

    ENTRY


Start    PROC  ; Начало программы

; Первая строчка кода на ассемблере

loop           ; Бесконечный цикл
    B loop
    ENDP       ; Конец программы
    END        ; Конец файла
```

2.9. Открываем вкладку «Flash / Configure Flash Tool». В открывшемся окне во вкладке «Linker» ставим галочку в строке «Use Memory Layout From Target Dialog» и нажимаем «ОК».

2.10. Открываем вкладку «Project» и нажимаем кнопку «Build Target». В окне статуса «Build Output» должны появится следующие строки:

```
Program Size: Code=12 RO-data=0 RW-data=0 ZI-data=0
Finished: 0 information, 1 warning and 0 error messages.
".\Objects\empty.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:01
```

0 Error(s) – Означает, что программа скомпилировалась корректно.

2.11. Горячие клавиши

Компиляция кода – F7.

Начать отладку – CTRL+F5.

3. Язык Ассемблера

Исходный текст программы на языке ассемблера имеет определенный формат. Каждая команда и директива представляет собой строку:

МЕТКА ОПЕРАЦИЯ ОПЕРАНД(Ы) КОММЕНТАРИИ

Поля могут отделяться друг от друга произвольным числом пробелов и табуляцией.

МЕТКА. В поле метки размещается символическое имя ячейки памяти, в которой хранится отмеченная команда или операнд. Метка представляет собой буквенно-цифровую комбинацию, начинающуюся с буквы. Используются только буквы латинского алфавита. В качестве символических имен и меток не могут быть использованы мнемокоды команд, директив и операторов ассемблера, зарезервированные имена, а также мнемонические обозначения регистров и других внутренних блоков микроконтроллера.

ОПЕРАЦИЯ. В поле операции записывается мнемоническое обозначение команды или директивы ассемблера, которое является сокращением (аббревиатурой) полного английского наименования выполняемого действия.

Например, MOV – move – переместить, JMP – jump – перейти, DB – define byte – определить байт. Для различных типов микроконтроллеров используется различный набор мнемонических кодов. Если компилятор обнаружит не известную микроконтроллеру операцию, то она будет восприниматься, как ошибочная.

ОПЕРАНД. В этом поле определяются операнды (или операнд), участвующие в операции. Команды ассемблера могут быть без-, одно- или двухоперандными. Операнды разделяются запятой **(,)**. Операнд может быть задан непосредственно или в виде его адреса (прямого или косвенного). Непосредственный операнд представляется числом **(MOV R0, #15)** или символическим именем **(ADD R0, #OPER2)** с обязательным указателем префикса непосредственного операнда **(#)**. Прямой адрес операнда может быть задан мнемоническим обозначением **(IN R0, P1)**, числом **(INC 40)**, символическим именем **(MOV R0, MEMORY)**. Указанием на косвенную адресацию служит префикс **@**. В командах передачи управления операндом может являться число **(LCALL 0135H)**, метка **(B LABEL)**, косвенный адрес **(V @ R0)** или выражение **(B $ - 2, где $ - текущее содержимое счётчика команд)**. Используемые в качестве операндов символические имена и метки должны быть определены, а числа представлены с указанием системы счисления, для чего используется суффикс (буква, стоящая после числа): B – для двоичной, Q – для восьмеричной, D – для десятичной и H – для шестнадцатиричной. Число без суффикса по умолчанию считается десятичным. Ассемблер допускает использование выражений в поле операндов, значения которых вычисляются в процессе трансляции. Выражение представляет собой совокупность символических имен и чисел, связанных операторами ассемблера. Операторы ассемблера обеспечивают выполнение арифметических (“+” – сложение, “-” – вычитание, “*” – умножение, “/” – целое деление, MOD – деление по модулю) и логических (OR – ИЛИ, AND – И, XOR – исключающее ИЛИ, NOT – отрицание) операций в формате 2-байтных слов. Например, запись **ADD R0, #((NOT 13)+1)** эквивалентна записи **ADD R0, #0F3H** и обеспечивает сложение содержимого аккумулятора с числом -13, представленным в дополнительном коде. Широко используются также операторы LOW и HIGH, позволяющие вычислить младший и старший байты 2-байтного операнда.

КОММЕНТАРИЙ. Поле комментария может быть использовано программистом для текстового или символьного пояснения логической организации прикладной программы. Поле комментария полностью игнорируется ассемблером, а потому в нём допустимо использовать любые символы. По правилам языка ассемблера поле комментария начинается с точки с запятой (;).

3.1. Директивы препроцессора

Директива **PRESERVE8** указывает, что текущий файл сохраняет восьми байтовое выравнивание стека.

Директива **AREA** инструктирует ассемблер собирать новый сегмент кода или данных. Для выполнения работы необходимо использовать следующие сегменты:

AREA RESET, DATA, READONLY — стартовый сегмент (стартап);

AREA |.text|, CODE, READONLY — программный сегмент;

AREA |.text|, DATA, READONLY — сегмент данных (FLASH);

AREA |data|, DATA, READWRITE — сегмент данных (RAM);

Директива **THUMB** инструктирует ассемблер интерпретировать последующие инструкции, как инструкции Thumb, используя синтаксис UAL.

Директива **DCD** выделяет одно или несколько слов памяти, выровненных по четырехбайтовым границам, и определяет начальное содержимое памяти во время выполнения.

Директива **EQU** дает символическое имя для числовой константы, относительного значения регистра или относительного значения PC регистра.

Директива **ENTRY** объявляет точку входа в программу.

Директива **GET** вставляет код из файла.

Директива **SPACE** резервирует места в памяти на 10 указанное число байт после директивы

3.2. Структура кода на языке ассемблера

Структура кода на языке ассемблера имеет сегментированный вид. Перед объявлением констант, переменных, кода или начала программы необходимо объявить определённый сегмент с помощью директивы AREA (п.2.12.3.). Перед написанием программы необходимо:

1. Создать startup.s файл, в котором будут инициализированы вектора прерываний и объявлена точка входа в программу. После инициализации необходимо сделать безусловный переход в основную программу main, где будет написан код для выполнения поставленной задачи.
2. Создать main.s файл в котором будет написана основная программа.
3. Создать файл stm32_EQU.s для описания макроподстановок необходимых для упрощения кода и читабельности программы.

3.2.1. Структура Startup файла

startup.s

(Файл необходим для первичной настройки микроконтроллера и объявления векторов прерываний)

Структура	Пояснение

1. Объявление имён числовых констант и меток	
2. Объявление стартового сегмента	С сегмента RESET компилятор считывает вектора прерываний
3. Объявление таблицы векторов прерываний	Объявление векторов прерываний всегда должно происходить после объявления стартового сегмента и в определённом порядке согласно таблице векторов из документации на микроконтроллер.
4. Объявление точки входа в программу	
5. Инициализация МК	Инициализация необходима для настройки источника тактирования и интерфейса для программирования и т.п.
6. Переход в основную программу	Основная программа обычно называется main. В ней можно писать код необходимы для выполнения поставленной задачи.
7. Конец файла	

Пример пустого startup.s представлен в листинге 2:

Листинг 2. Пример файла startup.s.

```
        GET main.s              ; Вставка файла main.s

; Директива PRESERVE8 указывает, что текущий файл требует или сохраняет восьми
байтовое выравнивание стека
        PRESERVE8

; Директива THUMB инструктирует ассемблер интерпретировать последующие инструкции,
как инструкции Thumb используя синтаксис UAL.
        THUMB

; Объявляем стартовый сегмент кода
        AREA RESET, CODE, READONLY

        ; Таблица векторов прерываний
        DCD STACK_TOP           ; Указатель на вершину стека
        DCD startup             ; Вектор сброса

; Точка входа
         ENTRY


; Startup код
startup PROC    ; Начало startup кода
        ; Инициализация МК

        ; Переход в программу main
        B main

        ENDP    ; Конец программы
        END     ; Конец файла
```

3.2.2. Файл main.s

Пример файла main.s с программой для инкрементирования значения регистре R0 представлен в листинге 3.

Листинг 3. Файла main.s.

```
        GET stm32_EQU.s
; Объявляем сегмент констант
        AREA CONSTANT_FLASH, DATA, READONLY

; Здесь объявляются константы

; Объявляем сегмент переменных
        AREA VERIABLE_RAM, DATA, READWRITE

; Здесь объявляются переменные

; Объявляем сегмент кода
                AREA MAIN, CODE, READONLY
                THUMB
; Объявляем функцию main
main    PROC
                ; Здесь пишется ваша программа
                ; Пример программы
                MOV R0, #NULL   ; R0 = 0x00
                MOV R1, #ONE    ; R1 = 0x01

loop    ; метка с именем loop
                ADD R0, R1              ; R0 = R0+R1
                B loop                  ; Безусловный переход на метку loop

                ENDP    ; Конец функции main

                END             ; Конец файла
```

3.2.3. Заголовочного файла

В заголовочном файле описываются макроподстановки с помощью директивы EQU.

Структура :

```
ИМЯ     EQU      ЗНАЧЕНИЕ       ; Комментарий
```

Пример:

```
STACK   EQU      0x20000000     ; адрес стека
```

Макроподстановка выполняется на этапе компиляции и всякий раз, когда в тексте программы будет встречаться ИМЯ, после этого определения, оно будет заменяться на ЗНАЧЕНИЕ.

Пример stm32_EQU.s представлен в листинге 4:

```
; Определение макроподстановок

STACK           EQU     0x20000000              ; адрес стека
STACK_SIZE      EQU     0x500                   ; размер стэка
STACK_TOP       EQU     STACK+STACK_SIZE        ; вершина стека


NULL            EQU     0x00
ONE             EQU     0x01
        END
```

3.2.4. Пример выполнения лабораторной работы.

Задание: дан одномерный массив, состоящий из 10 однобайтных чисел. Найти количество нулей в массиве и записать результат в ОЗУ.

1) Для начала необходимо создать проект и написать шаблон согласно п.3.2.

2) Необходимо ознакомится с заданием и составить блок схему программы (Блок схема программы составляется в программе Visio). Блок схема для примера представлена на рисунке 8.

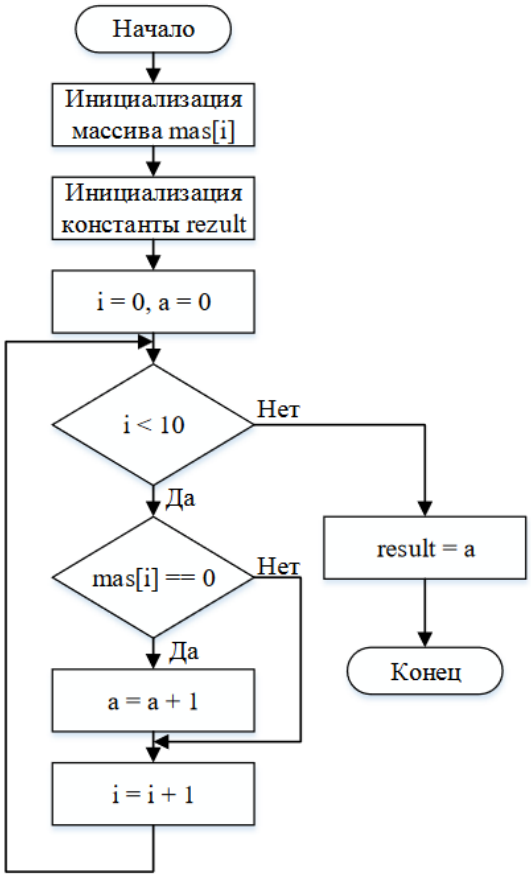

Рис 8. Блок схема программы.

3) Исходя из блок схемы программы видно, что для начала необходимо проинициализировать массив, состоящий из 10 однобайтных чисел и переменную, в которой будет храниться результат вычисления.

Объявление массива констант происходит в сегменте данных во FLASH памяти, так как не требуется изменять значения массива. Объявление переменной, в которой будет сохраняться результат выполнения задания, объявляется в сегменте данных в ОЗУ (RAM), так как это значение необходимо менять входе выполнения программы.

Для инициализации массива необходимо с первого символа строки написать метку mas (имя массива), затем через пробел написать директиву размера DCB и через запятую перечислить 10 однобайтных чисел. Пример:

```
; Объявляем сегмент констант
        AREA    CONSTANT_FLASH, DATA, READONLY
```

```
mas     DCB     0x11, 0x22, 0x33 …      ; Инициализация массива
```

Для инициализации переменной необходимо с первого символа строки написать метку result (имя переменной), а затем через пробел директиву SPACE и число. Директива SPACE резервирует место в памяти указанное число байт после директивы.

```
; Объявляем сегмент переменных
        AREA VERIABLE_RAM, DATA, READWRITE
result  SPACE   1       ; Инициализация переменной размером в 1 байт
```

4) Далее в блок схеме программы указаны действия обнуления переменных **i** и **a**. Отметим для себя что переменная **i** это регистр общего назначения **R0**, а переменная **a** это регистр **R1**. Для обнуления регистров можно использовать инструкцию MOVW или воспользоваться логическим операцией EOR.

5) После обнуления переменных идет проверка условия **i < 10** .Организация условия включает в себя использование меток, инструкции CMP и инструкции условного перехода.

6) Если условие «истина», то переходим к проверке элемента массива. Для обращения к массиву необходимо с помощью инструкции LDRB узнать адрес первого элемента массива. Затем считать значение из памяти в регистр и снова организовать условие. Если «ложь» устанавливаем переход на конец программы.

7) При выполнении условия регистр **R0** и **R1** складываются с единицей, в противоположном случае складывается с единицей только **R0**.

8) Конечным кодом программы является организация цикла для обработки всех элементов массива и при выходе из цикла запись результата переменную result.

9) По окончанию написания кода необходимо зайти под отладчиком и проверить правильность работы программы.

10) Убедившись правильности выполнения программного кода поставленной задачи, необходимо оформить отчет по лабораторной работе согласно СТО 4.2-07-2014. И подготовить ответы на вопросы к лабораторной работе.

В листинге 4 представлен код программы для выполнения задания из п.3.2.4.

Листинг 3. Файл main.s с кодом для выполнения задания из п.3.2.4.

```
GET stm32_EQU.s
; Объявляем макроподстановку для максимального размера массива
MAX_SIZE        EQU     0xA
; Объявляем сегмент констант
        AREA CONSTANT_FLASH, DATA, READONLY
; Здесь объявляются константы
; Инициализация массива mass с произвольными значениями
mas DCB 0x01, 0x00, 0x02, 0x00, 0x03, 0x00, 0x04, 0x00, 0x05, 0x00

; Объявляем сегмент переменных
        AREA VERIABLE_RAM, DATA, READWRITE
; Здесь объявляются переменные

; Инициализация переменной, в которой будет храниться результат
result SPACE 0x01

; Объявляем сегмент кода
                AREA MAIN, CODE, READONLY
```

```
                THUMB
; Объявляем функцию main
main    PROC
                ; Обнуляем регистры, в которых будем хранить переменные R0 = "i" и
R1 = "a"
                MOV R0, #NULL           ; i = R0 = 0x00
                EOR R1, R1              ; a = R1 = 0x00
                ; Запись адреса первого элемента массива в регистр "R2"
                LDR R2, =mas

CHECK   ; Метка CHECK
                ; Проверка "i" > "MAX_SIZE"
                CMP R0, #MAX_SIZE
                ; Если i < MAX_SIZE, переходим в "CALC"
                BLT CALC1
                ; Безусловный переход в "EXIT". Выполняется только, когда не
произошёл переход в "CALC1"
                B       EXIT
CALC1   ; Метка CALC1

                ; Запись в регистр "R3" значения по адресу сохранённом в "R2" плюс
смещение в регистре "R0".
                LDRB R3, [ R2, R0]      ; R3 = mas[i]
                ADD R0, #ONE            ; R0 = R0 + 1
                ; Сравнение считанного значения с NULL
                CMP R3, #NULL
                ; Если R3 != NULL, то переходим в CHECK
                BNE CHECK               ; Условный переход в CHECK
                ADD R1, #ONE ; a = a + 1
                ; Безусловный переход в "CHECK"
                B CHECK
EXIT    ; Метка EXIT
                ; Запись адреса переменной "result" в регистр "R2"
                LDR R2, =result
                ; Запись значения регистра R1 в память по адресу записанного в
регистре R2
                STRB R1, [R2]           ; result = a
                ENDP                    ; Конец функции main
                END                     ; Конец файла
```

4. Варианты заданий

Дан одномерный массив, состоящий из 10 однобайтных чисел, обработать массив согласно заданию:

1. Написать программу расчета среднего арифметического значения положительных элементов в одномерном массиве, имеющих четные индексы.
2. Написать программу вычисления суммы отрицательных, произведения положительных и количества нулевых значений в одномерном массиве.
3. Написать программу расчета суммы положительных элементов одномерного массива, имеющих нечетные индексы.
4. Упорядочить одномерный массив в порядке убывания.
5. Написать программу расчета среднего арифметического значения отрицательных элементов в одномерном массиве. Заменить минимальный элемент в одномерном массиве на среднее арифметическое.

6. Упорядочить одномерный массив в порядке возрастания.

7. Сформировать массив [Ai] из элементов одномерного массива [Bi] по закону:

$$A_i = (B_i + B_{N-i+1})/4,\ i = \overline{1,N}$$

8. В одномерном массиве поменять местами максимальный и минимальный элементы.

9. Написать программу расчета произведения положительных элементов в одномерном массиве. Заменить максимальный элемент в одномерном массиве на произведение.

10. Отыскать последний положительный элемент в одномерном массиве и заменить его на среднее арифметическое элементов массива.

11. Из одномерного массива [Ai]сформировать одномерный массив [Bi], записав в него сначала элементы массива A, имеющие четные индексы, потом –элементы с нечетными индексами.

12. Отыскать последний отрицательный элемент в одномерном массиве и заменить его на произведение элементов массива.

13. Заменить в одномерном массиве нулевые элементы на значение минимального элемента.

14. Сформировать массив [Xi], элементы которого равны частоте встречаемости элементов массива [Bi] среди элементов массива [Ai] Определить, какой элемент массива [Bi] чаще всего встречается в [Ai].

15. Сформировать массив [Xi], элементы которого равны полу сумме двух соседних элементов одномерного массива [Yi].

5. Литература для подробного изучения материала

6. Вопросы для защиты

1. Системы счисления (двоичная, восьмеричная, десятичная, шестнадцатеричная), перевод из одной системы в другую.
2. Логические элементы (И, ИЛИ, НЕ, ИЛИ-НЕ, И-НЕ, ИСКЛЮЧАЮЩИЕ-ИЛИ).
3. Триггеры (асинхронные-RS, синхронные-RS, динамические-DT,статические-DT, TT).
4. Общая структурная схема микропроцессора.
5. Стек, РОНы.
6. Регистрфлагов
7. Инструкции пересылки.
8. Инструкции перехода.
9. Арифметические инструкции.
10. Инструкции сдвига.
11. Логические инструкции

Лабораторная работа №2. Изучение функций и процедур.

1. Вызовы функций

В языках высокого уровня поддерживаются функции (их также называют процедурами, или подпрограммами) для повторного использования часто выполняемого кода и для того, чтобы сделать программу модульной и удобочитаемой. У функций есть входные данные, называемые аргументами, и выходной результат, называемый возвращаемым значением. Функция должна вычислять возвращаемое значение, не вызывая неожиданных побочных эффектов.

Когда одна функция вызывает другую, вызывающая и вызываемая функции должны согласовать, где размещать аргументы и возвращаемое значение. В архитектуре ARM принято соглашение о том, что вызывающая функция размещает до четырех аргументов в регистрах R0–R3, перед тем, как произвести вызов, а вызываемая функция помещает возвращаемое значение в регистр R0, перед тем как вернуть управление. Следуя этому соглашению, обе функции знают, где искать аргументы и возвращенное значение, даже если вызывающая и вызываемая функции были написаны разными людьми.

В случаях, когда необходимо передать в функцию значения над которыми требуется произвести, какие-либо действия, то следует использовать регистры общего назначения для передачи их в функцию. Если вызываемая программа написана на языке Си, то компилятор сгенерирует инструкции для сохранения используемых в функции регистров для последующего восстановления после выполнения функции. Передача параметров в функцию происходит по стандарту вызовов AAPCS (Procedure Call Standard for the Arm Architecture). В данном стандарте определено функциональное назначение РОН. Функциональное назначение регистров в соответствии со стандартом AAPCS приведено в таблице 5.

Таблица 5. Функциональное назначение регистров в соответствии со стандартом AAPCS.

РОН	Функциональное имя	Специальное назначение	Пояснение
R0	a1		Аргумент №1 Возвращаемое значение №1
R1	a2		Аргумент №2 Возвращаемое значение №2
R2	a3		Аргумент №3
R3	a4		Аргумент №4
R4	v1		Временная переменная №1
R5	v2		Временная переменная №2
R6	v3		Временная переменная №3
R7	v4		Временная переменная №4
R8	v5		Временная переменная №5
R9	v6		Временная переменная №6
R10	v7		Временная переменная №7
R11	v8	FP	Указатель на кадр

			Временная переменная №8
R12		IP	Регистр внутри процедурного вызова
R13		SP	Указатель стека
R14		LR	Адрес возврата
R15		PC	Счётчик команд

В соответствии со стандартом AAPCS, вызывая функцию, представленную в листинге 4, программе необходимо перед переходом в функцию положить число «**a**» в регистр «R0», число «b» в регистр «**R1**». После вычисления суммы «**a**» и «**b**» результат должен лежать в регистре «**R0**». После возврата из функции **sum ()**, программа запишет результат, лежащий в регистре «**R0**», в память по адресу переменной «**c**».

Листинг 4. Вызов функции sum().

```
c = sum(a,b);    //  c = a + b;
```

Вызываемая функция не должна вмешиваться в работу вызывающей. Это означает, что вызываемая функция должна знать, куда передать управление после завершения работы, и не должна изменять значения регистров или памяти, необходимых вызывающей функции. Вызывающая функция сохраняет адрес возврата в регистре связи LR одновременно с передачей управления вызываемой функции путем выполнения команды перейти и связать (branch and link, BL). Вызываемая функция не должна изменять архитектурное состояние и содержимое памяти, от которых зависит вызывающая функция. В частности, вызываемая функция должна оставить неизменным содержимое сохраняемых регистров (R4–R11 и LR) и стека – участка памяти, используемого для хранения временных переменных. В этом разделе мы покажем, как вызывать функции и возвращаться из них. Мы увидим, как функции получают доступ к входным аргументам и возвращают значение, а также то, как они используют стек для хранения временных переменных.

В архитектуре ARM для вызова функции используется команда *перейти и связать* (`BL`), а для возврата из функции нужно поместить регистр связи в счетчик команд (`MOV PC, LR`). Команды BL (перейти и связать) и MOV PC, LR необходимы для вызова и возврата из функции. BL выполняет две операции: сохраняет адрес следующей за ней команды (адрес возврата) в регистре связи (LR) и производит переход по указанному адресу.

2. Стек. Загрузка данных в стек и извлечение данных из стека.

Под стеком понимается определённая модель использования памяти. Вообще говоря, стеком является просто некий участок системной памяти, который за счёт использования регистра указателя (SP), имеющегося в процессоре, работает как буфер LIFO (англ. Last In, First Out – «последним пришёл — первым ушёл»). Чаще всего стек задействуется для сохранения содержимого регистров перед выполнением каких-либо операций над данными и последующего

восстановления содержимого этих регистров по окончании обработки данных.

В процессоре Cortex-M3 для работы со стеком предусмотрены команды PUSH (перевод с англ. «толкать») (загрузка регистров в стек) и POP (перевод с англ. «тянуть») (извлечение регистров из стека). Пример использования команд PUSH и POP приведены в листинге 4.

Листинг 5. Пример использования стека. Загрузка и выгрузка регистра R0.

```
                PUSH {R0} ; SP = SP -4, затем запись по адресу в регистре SP
значения из регистра R0
                ; Если i < MAX_SIZE, переходим в "CALC"
                POP {R0} ; запись в регистр R0 значения адресу лежащим в регистре
SP, затем SP = SP-4
```

Пример работы стека представлен на рисунке 8.

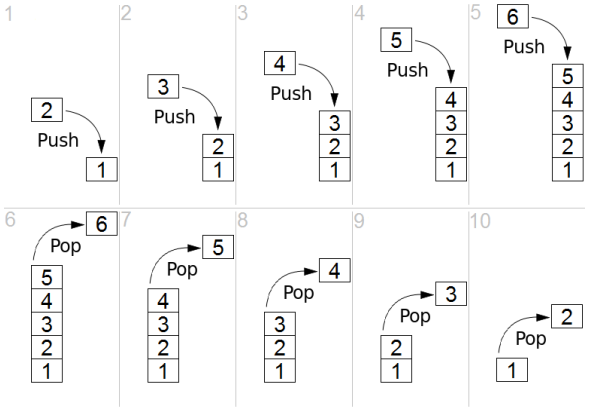

Рис 8. Пример работы стека.

При выполнении операций загрузки в стек и извлечения стека регистр указателя, обычно называемый указателем стека, автоматически изменяется таким образом, чтобы последующие операции со стеком не повредили ранее сохранённые данные.

В процессоре Cortex-M3 реализован «полный» убывающий стек (более подробно об этом можно узнать из книги «Ядро Cortex-M3. Полное руководство» раздела 3.6 «Основные стековые операции»). Соответственно, указатель стека при помещении нового значения в стек декрементируется. Команды PUSH и POP обычно используется для сохранения содержимого этих регистров из стека перед выходом из подпрограммы. С помощью одной команды можно загрузить в стек или восстановить из стека сразу несколько регистров (листинг 6).

Листинг 6. Пример использования стека. Загрузка и выгрузка нескольких регистров.

```
                PUSH {R0-R7, R12, R14}; Загрузить в стек
                POP  {R0-R7, R12, R14}; Выгрузить
```

3. Варианты заданий

a. Выполнить задание лабораторной №1 на языке Си используя функции и указатели.

b. Переписать программу, написанную в лабораторной работе №1 с помощью процедур языка ассемблера.

c. Вызвать процедуры, написанные на ассемблере в коде на языке Си.

d. Сравнить результат выполнения функций на языке Си и на языке ассемблера.

При выполнении работы необходимо использовать startup.s файл представленный в листинге 5.

Листинг 5. Файл startup.s с кодом для выполнения лабораторной работы №2.

```
Stack_Size      EQU     0x400
        AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>
Heap_Size       EQU     0x200
                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
                PRESERVE8
                THUMB
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                        DCD     __initial_sp   ; Top of Stack
                        DCD     Reset_Handler  ; Reset Handler
                AREA    |.text|, CODE, READONLY

; Reset handler
Reset_Handler    PROC
     IMPORT  __main
                LDR     R0, =__main
                BX      R0
; B main_asm
                ENDP
                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap

__user_initial_stackheap
                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + Stack_Size)
                LDR     R2, = (Heap_Mem +  Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR

                ALIGN

                END
               ; Запись адреса переменной "result" в регистр "R2"
               LDR R2, =result
               ; Запись значения регистра R1 в память по адресу записанного в
регистре R2
               STRB R1, [R2]          ; result = a
               ENDP                   ; Конец функции main
               END                    ; Конец файла
```

4. Пример выполнения лабораторной работы.

Задание: сформировать массив array[], в котором элементы равны сумме всех элементов массива array_original[] разделённые на номер элемента массива.

Листинг 6. Файл main.c с кодом для выполнения лабораторной работы №2.

```
#include <stdio.h>              /* Заголовочный файл объявляет несколько
целочисленных типов и макросов */
```

```c
#include <stdint.h>            /* Заголовочный файл объявляет несколько
целочисленных типов и макросов */
/*
Задание:
Сформировать массив array[], в котором элементы равны сумме всех элементов массива
array_original[] разделенные на номер элемента массива.
*/
// Инициализируем первичный массив с числами
int32_t array_original [] = {68,-69,28,27,-62,58,-50,49,-75,96,-19,35,41,
        -79,52,86,55,82,71,90,-88,45,-94,-59,3,80,38,-87,57,-48,89,-15,-54,-40,
        97,16,98,-53,7,64,51,66,46,63,-24,37,-70,22,-78,67};

// Инициализируем указатель на новый массив. Начальное значение NULL
int32_t* array = NULL;

// Объявляем прототип функции, которая будет выполнять задание
int32_t* array_treatment (int32_t* array_ptr, size_t array_size);

// Объявляем прототип функции, которая будет вычислять сумму элементов массива
int32_t array_sum (int32_t* array_ptr, size_t array_size);

//Экспортируем функцию из main.s написанную на ассемблере.
extern void array_treatment_asm (int32_t* array_ptr, size_t array_size, int32_t*
new_array_ptr );
extern int32_t array_sum_asm (int32_t* array_ptr, size_t array_size);

// Объявляем мпустой массив для передачи его адреса в функцию array_treatment_asm
int32_t new_array [sizeof(array_original)/sizeof(int32_t)]= {0};

int main (void)
{
/*Вызываем функцию array_treatment.
Функция возвратит указатель на массив соответствующий заданию.
Размерность массива такая же как у оригинального массива.*/

        array = array_treatment(array_original,
sizeof(array_original)/sizeof(int32_t));

/*Вызываем функцию array_treatment_asm, тело которой описано в файле main_1.s.
Так как, в языке ассемблера не функций malloc() и вообще нет механизма выделения
данных из кучи, то мы заранее выделяем место в оперативной памяти для
обработанного массива, поэтому нам необходимо передать адрес нового массива в
функцию. Размерность массива такая же как у оригинального массива.*/

        array_treatment_asm(array_original,
sizeof(array_original)/sizeof(int32_t), new_array);

        while (1)
        {
        }
        return 0;
}

/*****************************************************************************/
/*Функция формирующая массив array[], в котором элементы равны сумме всех
элементов массива array_original[] разделенные на номер элемента массива.*/
/*****************************************************************************/
/*
Функция принимает:
        int32_t* array_ptr - указатель на исходный массив с элементами типа
int32_t
        array_size                              - размер массива
Функция возвращает:
        указатель на область памяти в котором лежит новый массив
*/
```

```
int32_t* array_treatment (int32_t* array_ptr, size_t array_size)
{
	int32_t *new_array = NULL;
	int32_t sum = 0;

	//Выделяем область памяти для нового массива из кучи (heap).
	new_array =  (int32_t*) (malloc(array_size));

	// Вычисляем сумму элементов массива
	sum = array_sum(array_ptr, array_size);

	// При выполнении задачи происходит деление на ноль при заполнении первого
элемента. Поэтому мы запишем в нулевой элемент массива максимальное число типа
int32_t
	*(new_array+0) = ((uint32_t)(~0)) /2;

	for (uint32_t i = 1; i < array_size; i++)
	{
		*(new_array+i) = sum/i;
	}
	return new_array;
}

/*****************************************************************************/
/*Функция рассчитывает сумму элементов массива*/
/*****************************************************************************/
/*
Функция принимает:
int32_t* array_ptr - указатель на исходный массив с элементами типа int32_t
array_size	- размер массива
Функция возвращает:
сумму всех элементов массива
*/
int32_t array_sum (int32_t* array_ptr, size_t array_size)
{
	int32_t sum = 0;
	int32_t i = 0;

	while (i < array_size)
	{
		sum = sum + *(array_ptr+i);
		i++;
	}
	return sum;
}
```

Листинг 7. Файл main_1. s с кодом для выполнения лабораторной работы №2 на ассемблере.

```
   AREA |.text|, CODE, READONLY

array_sum_asm  PROC    ; Начало функции array_sum_asm
	EXPORT array_sum_asm
; Функция (процедура) реализует суммирование массива
; Функция принимает указатель на массив через регистр R0
; размер массива через регистр R1.
; Результат работы функции возвращается через регистр R0
; R0 = *array
; R1 = size(array)
; return R0= sum_array
	MOV R2, #0     ; В регистре R2 будем хранить номер элемента массиа
	MOV R3, #0     ; В регистре R3 будем хранить сумму элементов массиа
	MOV R4, #4     ; Размер одного элемента массива = 4 байтам,
	MUL R1, R4     ; поэтому умножаем кол-во элементов массива на 4.
sum
	CMP R2,R1      ; Проверка R2 > R1
```

```
        BLT calc        ; Если R2 < R1, переходим в "CALC"
        B exit          ; Если R2 > R1, выходим из программы
calc
        LDR R4, [R0,R2]; Считать содержимое элемента массива со смещением в R2
        ADD R3,R3,R4    ; R3 = R3 + R4
        ADD R2,#4
        B sum
exit
        MOV R0, R3
        BX LR
        ENDP    ; Конец функции array_sum_asm

array_treatment_asm    PROC    ; Начало функции array_treatment_asm
        EXPORT array_treatment_asm
; Функция (процедура) реализует обработку массива в соответствии с заданием
; Функция принимает указатель на массив через регистр R0,
; размер массива через регистр R1,
; указатель на пустой массив через регистр R2
; R0 = *array
; R1 = size(array)
; R2 = *new_array
        MOV R3, #0      ; В регистре R3 будем хранить номер элемента массиа
        MOV R4, #0      ; В регистре R4 будем хранить сумму элементов массиа

        ; Вызываем процедуру вычисления суммы массива
        PUSH {R0-R3, LR}        ; записываем все необходимые регистры в стек,
необходимые для процедуры array_treatment_asm
        ; в регистрах R0 и R1 уже лежит необходимые данные для вычисления суммы
        BL array_sum_asm
        MOV R4, R0                      ; запись суммы массива в регистр R4
        POP {R0-R3, LR}         ; возвращаем записанные ранее регистры из стека

        MOV R5, #4      ; Размер одного элемента массива = 4 байтам,
        MUL R1, R5      ; поэтому умножаем кол-во элементов массива на 4.

;При выполнении задачи происходит деление на ноль при заполнении первого элемента.
;Поэтому мы запишем в нулевой элемент массива максимальное число типа int32_t = 2
147 483 647
        MOV R5, #2147483647
        STR R5, [R2,R3]

        ADD R3,#4 ; смещаем относительный адрес массива

treatment
        CMP R3,R1               ; Проверка R3 > R1
        BLT treatment_calc      ; Если R2 < R1, переходим в "CALC"
        B exit_treatment        ; Если R2 > R1, выходим из программы

treatment_calc

        ASR R6, R3, #2
        SDIV R5, R4, R6
        STR R5, [R2,R3]
        ADD R3,#4               ; смещаем относительный адрес массива
        B treatment

exit_treatment
        BX LR   ; Возвращаемся назад по адресу расположенному в регистре LR
        ENDP    ; Конец функции array_treatment_asm

        END     ; Конец файла
```

5. Вопросы к лабораторной работе.

1. Как в языке Си объявляются переменные указателя на тип данных int?

2. Стандартные библиотеки языка Си.
3. Типы данных в языке Си.
4. Циклы в языке Си. Циклы на языке ассемблера.
5. Что такое стек? Инструкции для работы со стеком. Указатель стека.
6. Функциональное назначение РОН по стандарту AAPCS.
7. Регистр LR. Функциональное назначение.
8. Инструкции перехода с сохранением адреса возврата.
9. Что такое указатель?
10. Операции с указателями. Взять указатель, записать по указателю.
11. Директива препроцессора **#include**. Эквивалент директивы **#include** в языке ассемблера?
12. Директива препроцессора **#define**. Эквивалент директивы **#define** в языке ассемблера?
13. Значения адресов FLASH и SRAM памяти в микроконтроллере STM32F103C8?

6. Литература.

1. **Козаченко В.Ф. и др. - Практический курс микропроцессорной техники на базе процессорных ядер ARM-Cortex-M3 M4 M4F – 2019.**

Глава 14. Работа со стеком. Вложенные подпрограммы.
Глава 15. Программная реализация типовых алгоритмических структур.

2. **Джозеф Ю. Ядро cortex-m3 компании ARM.**

Глава 4.3.4. Язык ассемблера: вызов подпрограмм и безусловный переход.

3. **Брайан Керниган и Деннис Ритчи. Язык программирования Си.**

Глава 2. Типы данных, операции и выражения.
Глава 4. Функции и структура программы.
Глава 5. Указатели и массивы.

Лабораторная работа №3. Порты ввода-вывода (GPIO). Прерывания.

1. Порты ввода-вывода в микроконтроллерах STM32F1xx.

Порт ввода-вывода (GPIO с англ. general-purpose input/output pin) предназначен для взаимодействия с внешними сигналами от различных приложений, а также управление приложениями. В качестве приложений могут выступать светодиоды, реле, кнопки различные датчики и прочее. Каждому порту ввода вывода соответствуют 16 ножек микроконтроллера. Пользователю доступны вывод данных на порт и считывание данных с порта. Также порт может подключаться к другим модулям таким, как UART, USB, Ethernet и прочее. Подключение порта к другим модулям является альтернативной функцией порта ввода вывода.

На рисунке 3.1 представлена схема устройства одной ножки микроконтроллера STM32F1xx.

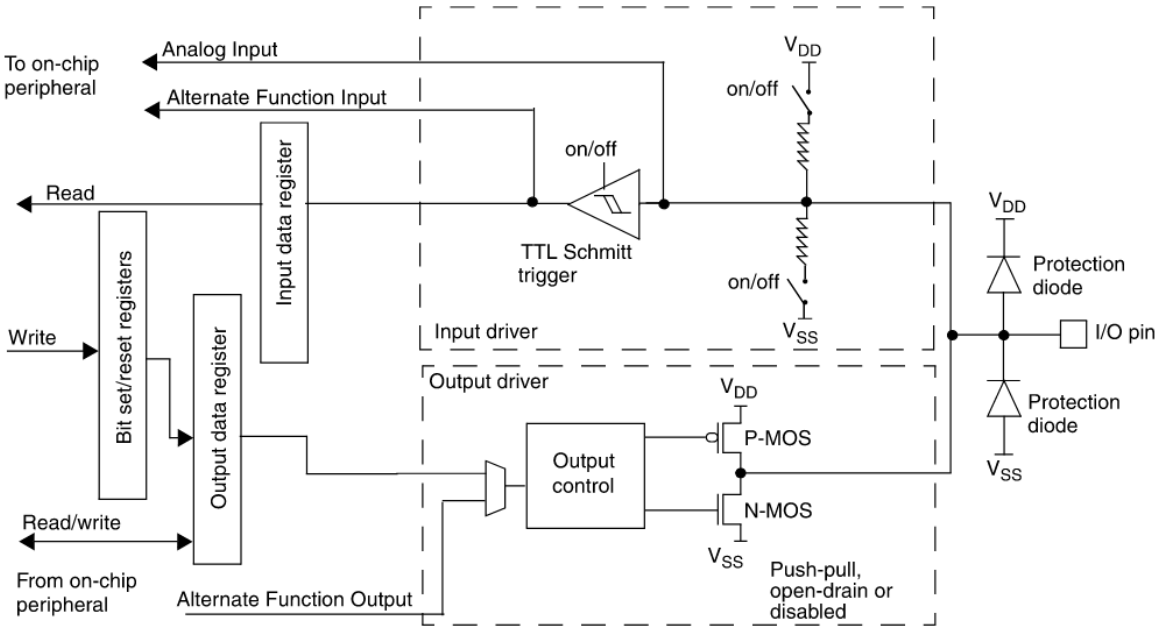

Рисунок 3.1. Изображение из Reference Manual, Figure 13. Basic structure of a standard I/O port bit

Порты ввода вывода имеют 8 возможных режимов работы (4 режима на вход и 4 режима на выход). Режимы работы на выход (рисунок 3.2):

- выход с открытым стоком (англ. output open-drain) — в этом случае при записи «0» в выходной регистр (англ. output register) активируется N-MOS, а при записи «1» порт переходит в высокоимпедансное состояние (также этот режим называют Hi-Z, P-MOS никогда не активируется);
- выход с подтяжкой «тяни-толкай» (англ. output push-pull) – в русской литературе называется «двухтактный выход» (запись «0» в Output register активирует N-MOS, запись «1» активирует P-MOS);
- альтернативная функция с подтяжкой «тяни-толкай» (англ. alternate function push-pull) – уже описанный ранее двухтактный выход, только для альтернативной функции;
- альтернативная функция с открытым стоком/коллектором (англ. alternate function open-drain).

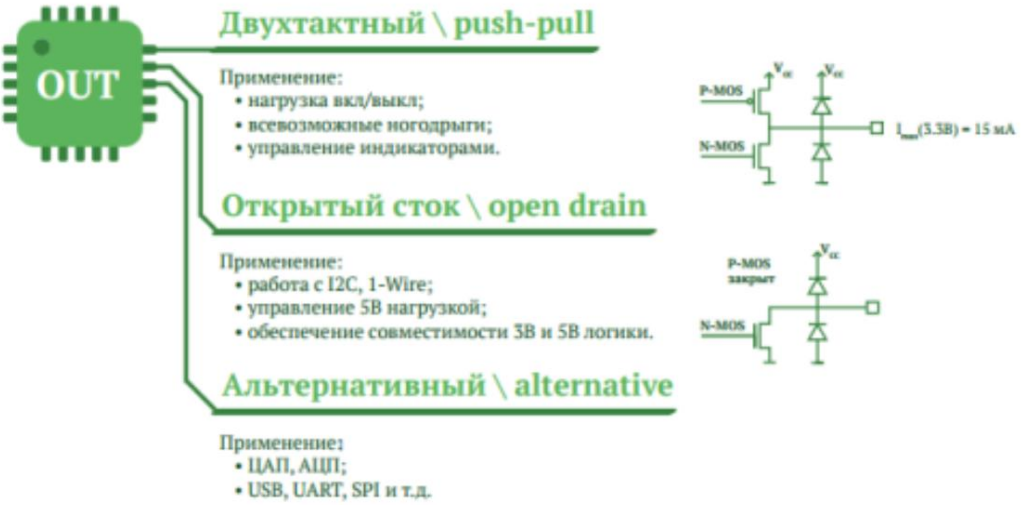

Рисунок 3.2. Функциональное назначение GPIO на выход

Режимы работы на вход:

- плавающий вход (англ. input floating) – подтяжка отключена (без подтяжки ножка находится в Hi-Z состоянии, а это означает, что сопротивление входа велико, и любая электрическая наводка (помеха) может вызвать появление на таком входе «1» или «0»);
- вход, подтянутый к верху (англ. input pull-up) — подтяжка к питанию (для STM32 обычно это 3,3 вольт);
- вход, подтянутый к низу (англ. Input-pull-down) — подтяжка к земле (0 вольт);
- аналоговый вход (англ. analog) — если ножка настроена как аналоговый вход, то используется модуль АЦП, если он подключен к этой ножке.

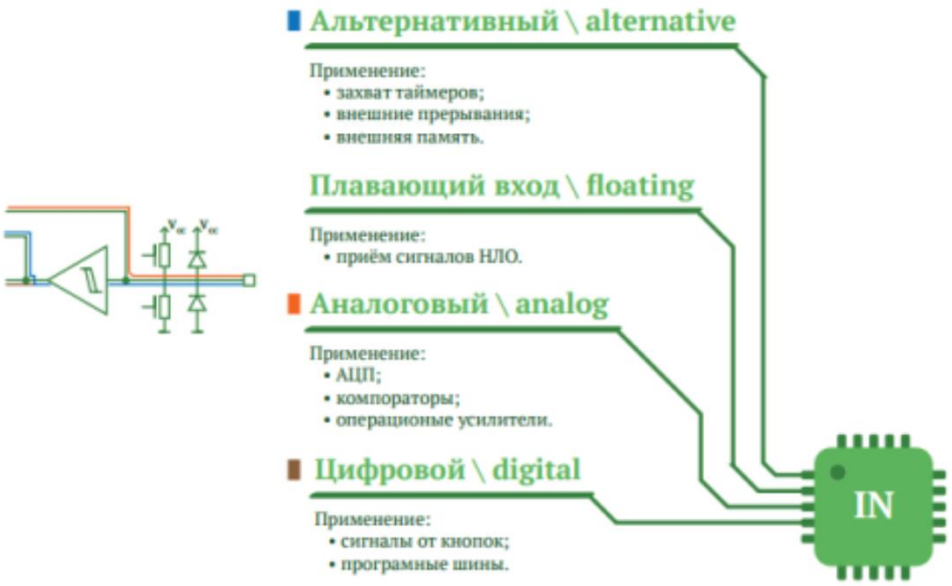

Рисунок 3.2. Функциональное назначение GPIO на вход

Для настройки модуля GPIO необходимо в первую очередь подключить тактирование с помощью модуля RCC (англ. Reset and Clock Control). После включения тактирования возможно изменение регистров модуля GPIO для настройки необходимого режима.

2. Система тактирования в микроконтроллерах STM32F1xx

Для работы всем контроллерам, процессорам, цифровым электронным устройствам необходим источник тактовых импульсов. Он может быть как внешний в виде RC цепочки,

керамического или кварцевого осциллятора, кварцевого генератора, также может находиться внутри контроллера. Микроконтроллер STM32F1xx имеет множество вариантов тактирования и обладает гибкими и широкими настройками управления тактированием своей периферии.

На рисунке 3.3 представлена схема тактирования микроконтроллера STM32F1xx.

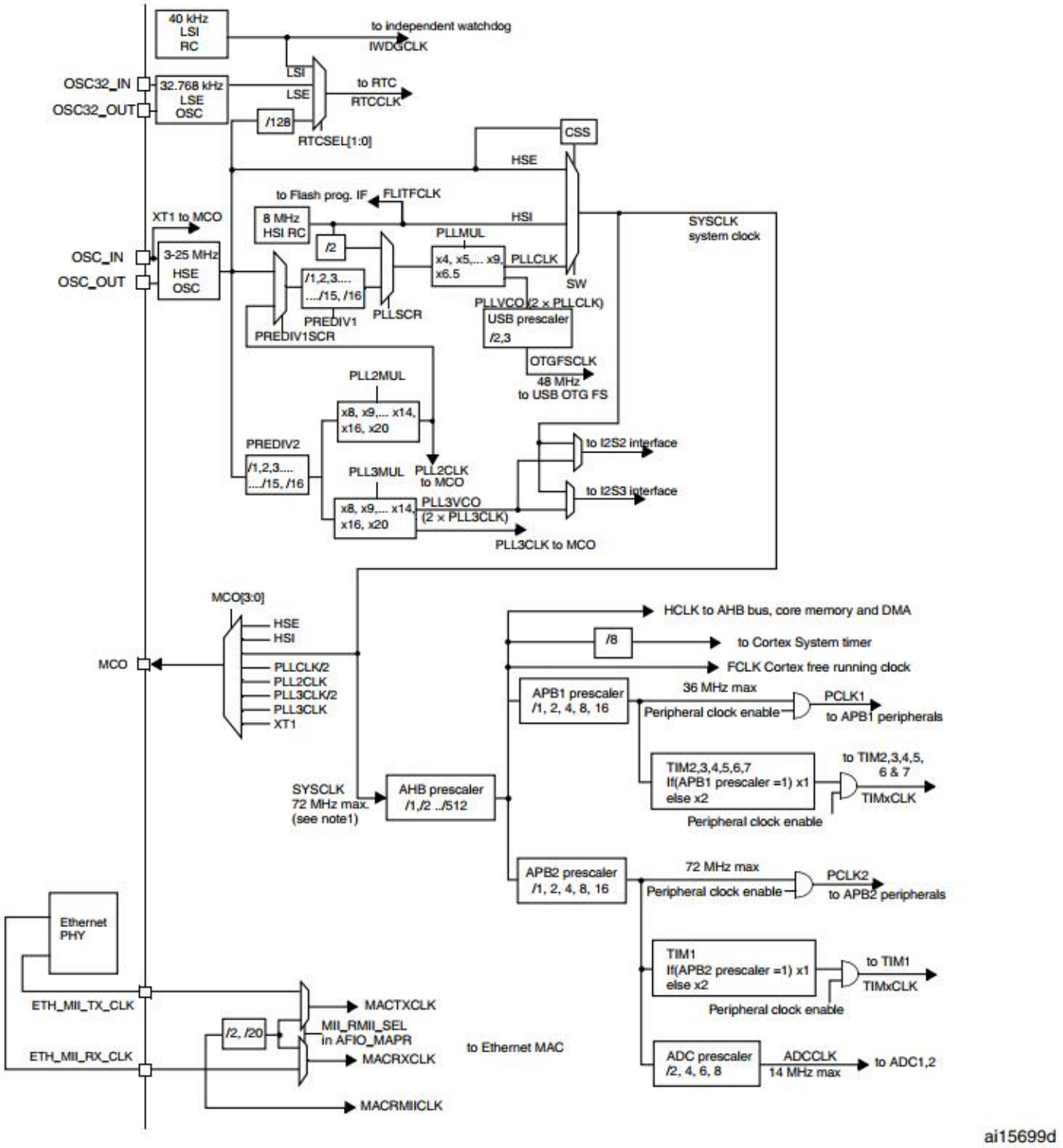

Рисунок 3.2. Изображение из Reference Manual, Figure 11. Clock tree

Из рисунка 3.3. видно, что три входа для внешнего генератора тактового сигнала. Вход LSE OSC32 (LSE OSC– Low Speed External oscillator) используется для тактирования модуля RTC (таймер реального времени). Вход HSE OSC (OSC– High Speed External oscillator) используется для тактирования ядра и периферии. Вход ETH_MII используется для

тактирования модуля интерфейса Ethernet. Помимо подключения внешних тактовых сигналов, внутри МК есть встроенные генераторы на основе RC цепочек. Генератор LSI OSC и генератор HSI OSC используются для тех же задач, что и внешнее генераторы, но имеют высокую нестабильность по сравнению с внешними генераторами на основе кварцевого резонатора.

Встроенные RC цепочки позволяют уменьшить количество компонентов в устройстве, меньше потребляют, чем при использовании внешнего источника тактового сигнала, но отличаются малой точностью и имеют температурный дрейф (те или иные отклонения в зависимости от окружающей температуры). Если устройство работает с точной высокоскоростной периферией, то использование встроенного генератора это не лучший вариант. Внешний источник тактовых сигналов отличается большей точностью и стабильностью, но при этом увеличивает количество компонентов, следовательно стоимость устройства и его габариты, но при этом, незаменим, когда требуется высокая точность и стабильность.

На макетной плате Nucleo-F103RB не установлен внешний тактовый генератор, поэтому необходимо пользоваться встроенным генератором HSI OSC. Модуль RCC по умолчанию настроен так, что тактовым сигналом является встроенный генератор HSI OSC с тактовой частотой 8 МГц. Для увеличения частоты тактирования от HSI OSC необходимо настроить соответствующие регистры модуля RCC управляющие умножителем частоты, делителями и выходным мультиплексором. На выходе мультиплексора тактовый сигнал называется SYSCLK, который используется для тактирования ядра микроконтроллера и является основой для тактирования периферии МК.

Тактовый сигнал SYSCLK поступает на блок делителя, с которого выходит тактовый сигнал шины AHB называемы HCLK. Тактовый сигнал шины AHB поступает на два делителя с которых выходят тактовые сигналы на шины APB1 и APB2. На рисунке 3.3 видно, что каждая шина подключена к определённой периферии и соответственно может иметь разные тактовые частоты.

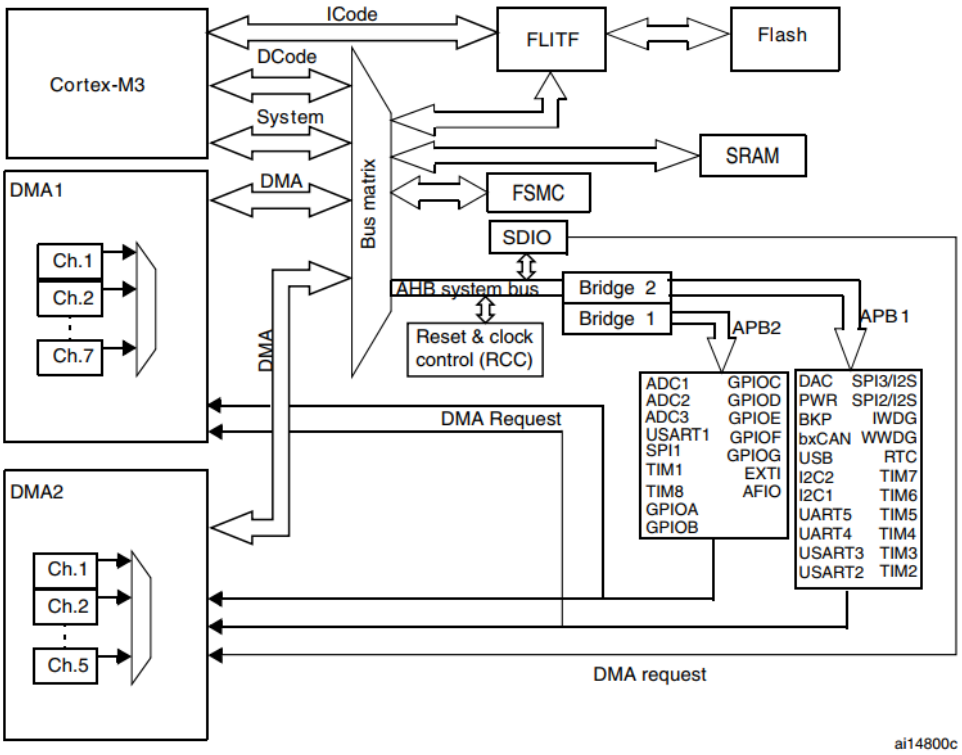

Рисунок 3.3. Изображение из Reference Manual, Figure 1. System architecture

Для подключения тактового сигнала к определённой периферии необходимо настроить регистры RCC_AHBENR (AHB Peripheral Clock enable register), RCC_APB2ENR (APB2 peripheral clock enable register), RCC_APB1ENR (APB1 peripheral clock enable register). Описание функционального назначения регистров и их битов приведено в документации на определённый

тип микроконтроллера. В случае с макетной платой «Nucleo-F103RB» основным документом, описывающим все регистры, является «Reference Manual RM0008».

3. Прерывания.

Прерывание (англ. interrupt) — сигнал от программного или аппаратного обеспечения, сообщающий процессору о наступлении какого-либо события, требующего немедленного внимания. Прерывание извещает процессор о наступлении высокоприоритетного события, требующего прерывания текущего кода, выполняемого процессором (https://ru.wikipedia.org/).

У STM32 прерываниями управляет контроллер прерываний NVIC — Nested vectored interrupt controller. NVIC представляет собой один из компонентов процессора Cortex-M3, тесно связанный с логикой ядра. Все регистры контроллера прерываний отображены на адресное пространство процессора. Помимо регистров управления обработкой прерываний и соответствующих узлов, контроллер NVIC также содержит регистры управления системным таймером SYSTICK и компонентами отладки.

Прерывания в процессорных системах делятся на системные и внешние. Системные прерывания, часто называют исключениями так, как они происходят от системных событий, например таких как сброс системы (Reset), ошибка чтения памяти (MemManage Fault), отказ шины (Bus Fault), и другие (Hard Fault, SYSTICK и т.п.). Внешние прерывания подключены к периферии микроконтроллера и генерируются по событию периферии. Например, при событии приёма байта данных от модуля UART может произойти прерывание, которое запустит соответствующий этому прерыванию обработчик.

Обработчик прерывания — это функция адрес, которой записан в таблицу векторов прерывания. Адреса обработчиков должны располагаться в соответствии с таблицей векторов т.е. в правильном порядке. В документации на микроконтроллер приведено описание каждой позиции в таблице векторов прерываний, также указан порядковый номер каждого обработчика (для платы NUCLEO-F103RB и микроконтроллера STM32F103 таблица векторов представлена в документе Reference Manual RM0008 на странице 198). Адреса обработчиков всегда расположены в стартап файле в области данных с пометкой «RESET». В листинге 3.1 представлена часть кода с заполнением таблицы векторов прерываний.

Функции обработчиков прерывания должны быть описаны в самой программе даже если они не понадобятся при работе программы, в противном случае необходимо удалить имена функций из таблицы. Обычно, для удобства, объявляют пустые обработчики прерываний в стартап файле с пометкой «[WEAK]» (с англ. слабый). Пометка «[WEAK]» позволяет переопределить функцию в случае, если в коде есть функция с точно таким же именем, но без метки «[WEAK]».

NVIC контроллер поддерживает от 1 до 240 входов внешних прерываний обычно называемыми входами запросов прерываний (IRQ). Точное число поддерживаемых прерываний определяется производителем конкретных микросхем на базе ядра Cortex-M3. Каждому внешнему прерыванию соответствует свой регистр уровня приоритета, разрядность которого в зависимости от конкретной реализации процессора может варьироваться от 3 до 8 бит. Общее количество регистров приоритета соответствует количеству прерываний.

Листинг 3.1 Часть startup.s файла с заполнением таблицы векторов прерываний.

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY

_Vectors         DCD     __initial_sp               ; Top of Stack
                 DCD     Reset_Handler              ; Reset Handler
                 DCD     NMI_Handler                ; NMI Handler
                 DCD     HardFault_Handler          ; Hard Fault Handler
                 DCD     MemManage_Handler          ; MPU Fault Handler
                 DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     SVC_Handler                ; SVCall Handler
                DCD     DebugMon_Handler           ; Debug Monitor Handler
                DCD     0                          ; Reserved
                DCD     PendSV_Handler             ; PendSV Handler
                DCD     SysTick_Handler            ; SysTick Handler

                 ; External Interrupts
                DCD     WWDG_IRQHandler            ; Window Watchdog
                DCD     PVD_IRQHandler             ; PVD through EXTI Line detect
                DCD     TAMPER_IRQHandler          ; Tamper
                DCD     RTC_IRQHandler             ; RTC
                DCD     FLASH_IRQHandler           ; Flash
                DCD     RCC_IRQHandler             ; RCC
                DCD     EXTI0_IRQHandler           ; EXTI Line 0
                DCD     EXTI1_IRQHandler           ; EXTI Line 1
                DCD     EXTI2_IRQHandler           ; EXTI Line 2
                DCD     EXTI3_IRQHandler           ; EXTI Line 3
                DCD     EXTI4_IRQHandler           ; EXTI Line 4
                DCD     DMA1_Channel1_IRQHandler   ; DMA1 Channel 1
                DCD     DMA1_Channel2_IRQHandler   ; DMA1 Channel 2
                DCD     DMA1_Channel3_IRQHandler   ; DMA1 Channel 3
                DCD     DMA1_Channel4_IRQHandler   ; DMA1 Channel 4
                DCD     DMA1_Channel5_IRQHandler   ; DMA1 Channel 5
                DCD     DMA1_Channel6_IRQHandler   ; DMA1 Channel 6
                DCD     DMA1_Channel7_IRQHandler   ; DMA1 Channel 7
                DCD     ADC1_2_IRQHandler          ; ADC1_2
                DCD     USB_HP_CAN1_TX_IRQHandler  ; USB High Priority or CAN1 TX
                DCD     USB_LP_CAN1_RX0_IRQHandler ; USB Low  Priority or CAN1 RX0
                DCD     CAN1_RX1_IRQHandler        ; CAN1 RX1
                DCD     CAN1_SCE_IRQHandler        ; CAN1 SCE
                DCD     EXTI9_5_IRQHandler         ; EXTI Line 9..5
                DCD     TIM1_BRK_IRQHandler        ; TIM1 Break
                DCD     TIM1_UP_IRQHandler         ; TIM1 Update
                DCD     TIM1_TRG_COM_IRQHandler    ; TIM1 Trigger and Commutation
                DCD     TIM1_CC_IRQHandler         ; TIM1 Capture Compare
                DCD     TIM2_IRQHandler            ; TIM2
                DCD     TIM3_IRQHandler            ; TIM3
                DCD     TIM4_IRQHandler            ; TIM4
                DCD     I2C1_EV_IRQHandler         ; I2C1 Event
                DCD     I2C1_ER_IRQHandler         ; I2C1 Error
                DCD     I2C2_EV_IRQHandler         ; I2C2 Event
                DCD     I2C2_ER_IRQHandler         ; I2C2 Error
                DCD     SPI1_IRQHandler            ; SPI1
                DCD     SPI2_IRQHandler            ; SPI2
                DCD     USART1_IRQHandler          ; USART1
                DCD     USART2_IRQHandler          ; USART2
                DCD     USART3_IRQHandler          ; USART3
                DCD     EXTI15_10_IRQHandler       ; EXTI Line 15..10
                DCD     RTC_Alarm_IRQHandler       ; RTC Alarm through EXTI Line
                DCD     USBWakeUp_IRQHandler       ; USB Wakeup from suspend
```

4. Использование прерываний

Для использования прерываний необходимо выполнить следующие операции:

- настроить стек;
- заполнить таблицу векторов прерываний;
- задать приоритеты прерываний;
- разрешить прерывания.

Для настройки стека требуется первым элементом таблицы векторов прерываний установить адрес стека, который автоматически запишется в регистор «SP».

Заполнение таблицы векторов рассмотрено в разделе 3.

Приоритеты прерываний задаются с помощью регистров контроллера прерываний NVIC таких, как PRI_0, PRI_1, PRI_2 и так далее до числа последнего прерывания.

Управление разрешением прерываний происходит в двух регистрах. Регистр NVIC_ISER предназначен для разрешения прерывания, а регистр NVIC_ICER для сброса. Регистры NVIC_ISER и NVIC_ICER являются 32-битными, каждый бит соответствует одному входу прерывания. Установка логической «1» в регистре NVIC _ICER автоматически сбрасывает соответствующий разряд регистра NVIC_ISER в логический «0». Благодаря такому решению (один регистр на установку, а другой на сброс) разрешение или запрещение прерывания не влияет на состояние битов разрешения остальных прерываний. В процессоре Cortex-M3 может быть больше 32 внешних прерываний, он может иметь несколько регистров NVIC_ISER и NVIC_ICER.

Поскольку первые 16 прерываний являются системными, внешнему прерыванию под номером «0», соответствует 16-ый бит в регистрах NVIC_ISER и NVIC_ICER.

После настройки контроллера NVIC, необходимо настроить периферию на генерацию сигнала прерываний, который соединён с контроллером NVIC. В случае модуля GPIO, необходимо в регистрах AFIO_EXTICR1…4 модуля AFIO настроить соответствующие биты для разрешения прерываний (AFIO это модуль для настройки альтернативных функций модуля GPIO). Установка соответствующего кода в регистры включит генерацию сигнала прерывания.

Заключительной настройкой прерыванiq для GPIO является настройка регистров модуля внешних прерываний EXTI.

Внимание! Перед записью в регистры модулей периферии (GPIO, AFIO, EXTI, UART и прочее) необходимо подключить к модулям тактовый сигнал с помощью модуля RCC. В противном случае запись регистров будет невозможна.

5. Контроллер внешний прерываний EXTI

Каналы EXTI имеют следующие названия: EXTI0, EXTI1, EXTI2 ... EXTI19. Всего в распоряжении 20 каналов. Причем EXTI0 — EXTI15 могут быть подключены к одному из портов GPIO. EXTI16 подключен внутри МК к выходу программируемого детектора напряжения модуля PVD, EXTI17 к событию модуля RTC Alarm, EXTI18 к модулю USB микроконтроллера, и EXTI19 к контроллеру Ethernet.

В данный момент нас интересуют те каналы EXTI, которые могут быть подключены к портам GPIO. Из рисунка 3.4 видно, что сигналы прерываний от портов GPIO объединены по разрядам через мультиплексор. А выходной сигнал с мультиплексора подключается к контроллеру прерываний. Это значит, что на все пины одинакового разряда разных портов имеют одно и тоже прерывание. Такая организация подключений имеет некоторое ограничение, которое необходимо учитывать: мы не можем одновременно регистрировать события, например, от линий PA0 и PB0, так как они подключены к одному и тому же мультиплексору.

Контроллер внешних прерываний имеет свои собственные регистры для управления прерываний. Включение прерываний задаётся с помощью регистра EXTI_IMR, в котором каждый бит соответствует номеру прерываний EXTI_0...19. С помощью регистров EXTI_RTSR и EXTI_FTSR настраивается детектор для генерации сигнала прерываний по нарастающему и/или спадающему фронту сигнала прерывания. Регистр EXTI_SWIER предназначен для генерирования программного прерывания (т.е. установка соответствующего бита в регистр

EXTI_SWIER сгенерирует сигнал прерывания для контроллера NVIC, а он в свою очередь запустит обработчик прерывания). Регистр EXTI_PR это регистр ожидания, каждый бит соответствует прерыванию, которое либо находится в ожидании, либо уже происходит. Данных бит не сбрасывается автоматически после выхода из прерывания.

Внимание! После возникновения прерывания необходимо сразу сбросить соответствующий прерыванию бит в регистре EXTI_PR, в противном случае после выхода из прерывания произойдёт повторное прерывание без события.

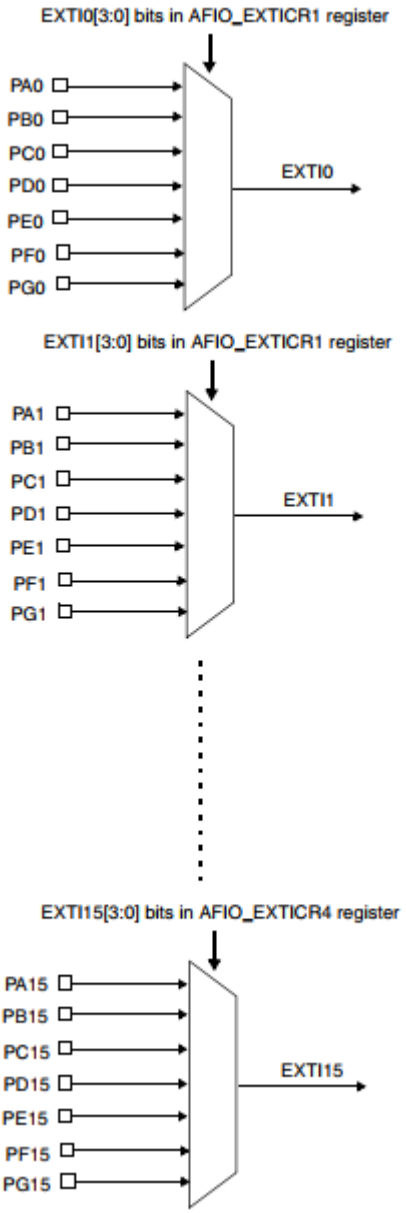

Рисунок 3.4. Изображение из Reference Manual, Figure 21. External interrupt/event GPIO mapping

В листенге 3.2 представлен пример инициализации прерывания по событию от порта C и пина 13.

Листинг 3.1 Настройка прерываний.

```
/*****************************************************************************/
/*Инициализация порта С пина 13 на вход для детектирования нажатия кнопки*/
/*****************************************************************************/

void init_gpio_port_C_pin_13 (void)
{
                /*Включение тактирования для порта С*/
                RCC->APB2ENR = RCC->APB2ENR | RCC_APB2ENR_IOPCEN_Msk;

                /*Настройка порта С пина 13*/
                GPIOC->CRH = (GPIOC->CRH) & ( ~( GPIO_CRH_MODE13_0 | GPIO_CRH_MODE13_1));
                GPIOC->CRH = ((GPIOC->CRH) | ( GPIO_CRH_CNF13_0 | GPIO_CRH_MODE13_1));
                GPIOC->ODR = ((GPIOC->ODR) | ( GPIO_PIN_13));
}

/*Включение прерываний от пина 13*/

void init_gpio_port_C_pin_13_interrupt (void)
{
                /*****************************************************************/
                /*Включение прерываний от пина 13*/
                /*****************************************************************/

                //Включение тактирования на блок альтернативных функций
                RCC->APB2ENR = RCC->APB2ENR | RCC_APB2ENR_AFIOEN;

                //Разрешить прерывание 13 пина порта С
                AFIO->EXTICR[3] = 0x0020;

                //Разрешить прерывание 13 линии
                EXTI-> IMR|=(EXTI_IMR_MR13);
                EXTI-> EMR|=(EXTI_EMR_MR13);

                //Прерывание 13 линии по спадающему фронту фронту
                EXTI->RTSR|=(EXTI_RTSR_TR13);

                /* Разрешение прерываний */
                // EXTI15_10_IRQn = 40 в соответствии с таблицей векторов прерываний
                NVIC->ISER[1] = (uint32_t)(1UL << (((uint32_t)EXTI15_10_IRQn) & 0x1FUL));
}

/*Обработчик прерывания*/
void EXTI15_10_IRQHandler (void)
{
                /*Сброс прерывания*/
                EXTI->PR |= GPIO_PIN_13;

                /*Здесь может быть ваша программа*/
                /**/
                /**/
                /**/
                /**/
}
```

6. Варианты заданий

1. Бегущие огни 0 -> 7.

Разработать программу бегущего огня (от разряда 0 к разряду 7) с настраиваемой частотой f = f0*(1 + CODE [0:7]). CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

2. Бегущие огни с реверсом.

Разработать программу бегущего светодиода (от 0 светодиода до 7 и от 7 до 0)с настраиваемой частотой f = f0*(1 + CODE[0:7]). CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

3. 8-битный инкрементирующий счётчик 0 -> 255.

Разработать программу восьмибитного счётчика (от 0 до 255) с выводом текущего значения в двоичной системе на светодиоды. Частота счётчика f = f0*(1 + CODE [0:7]) должна задаваться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

4. 8-битный декрементирующий счётчик 255 -> 0.

Разработать программу восьмибитного счётчика (от 255 до 0) с выводом текущего значения в двоичной системе на светодиоды. Частота счётчика f = f0*(1 + CODE [0:7]) должна задаваться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

5. Счётчик, инкрементирующий 8-битный с реверсом 0 -> 255 -> 255 -> 0.

Разработать программу восьмибитного счётчика (от 0 до 255 и от 255 до 0) с выводом текущего значения в двоичной системе на светодиоды. Частота счётчика f = f0*(1 + CODE [0:7]) должна задаваться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

6. 8-битный инкрементирующий счётчик 0 -> N.

Разработать программу восьмибитного счётчика (от 0 до N) с выводом текущего значения в двоичной системе на светодиод. Частота счётчика фиксированная (выбирается студентом). С помощью восьмибитного тумблера должно загружается максимальное значение счётчика. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка максимального значения счётчика (N) с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

7. Бегущая змейка 0 -> 7.

Разработать программу бегущего огня (от разряда 0 к разряду 7) с настраиваемым количеством бегущих огней. Количество бегущих светодиодов должно устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

8. Плавно пульсирующий и плавно гаснущий огонь.

Разработать программу плавно пульсирующих и плавно гаснущих светодиодов. Время увеличения и уменьшения яркости задаётся с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения с тумблера, сброс счётчика и запуск счётчика, при втором нажатии остановка счётчика.

9. Управляемый инкрементирующий двоичный счётчик N -> 0.

Разработать программу инкрементирующего восьмибитного счётчика с выводом текущего значения на светодиоды. Начальное значение счётчика должно задаваться с помощью тумблера и загружаться в программу по прерыванию от кнопки. При первом нажатии кнопки должна осуществляться загрузка начального значения с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

10. Кодовый замок.

Разработать программу кодового замка. Секретный код должен быть задан, как константа uint8_t. Код должен вводится с помощью тумблера. Загрузка значения с тумблера в программу должна осуществляться с помощью прерывания от кнопки. При успешном вводе кода должны загореться все 8 светодиодов на 5 секунд. При не корректном вводе все 8 светодиодов должны моргнуть 3 раза с периодом 1 секунда.

11. Гирлянда с эффектом "заполнение".

Разработать программу с последовательно зажигающимися светодиодами, затем гаснущими в той же последовательности. Частота счётчика f = f0*(1 + CODE [0:7]) должна быть настраиваемой. CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

12. 8-битный счётчик в коде Грея.

Разработать программу восьмибитного счётчика Грея с выводом текущего значения в двоичной системе на светодиод. Частота счётчика f = f0*(1 + CODE [0:7]) должна быть настраиваемой. CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

13. Зеркальный бегущий огонь.

Разработать программу двух зеркальных бегущих огней на двух четырёхбитных индикаторах. (Пояснение: Первые 4 светодиода загораются от 3 -> 0 последовательно и гаснут в обратной последовательности. Вторые 4 светодиода загораются от 4 -> 7 последовательно и гаснут в обратной последовательности). Частота изменения состояния светодиодов f = f0*(1 + CODE [0:7]) должна быть настраиваемой. CODE [0:7] должен устанавливаться с помощью восьми

битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

14. Плавно пульсирующий огонь.

Разработать программу плавно пульсирующих светодиодов. Время увеличения яркости задаётся с помощью восьми битного тумблера. Загрузка значения тублера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения с тумблера, сброс счётчика и запуск счётчика, при втором нажатии остановка счётчика.

15. 8-битный декрементирующий счётчик 255 -> 0.

Разработать программу восьмибитного счётчика (от 255 до 0) с выводом текущего значения в двоичной системе на светодиоды. Частота счётчика f = f0*(1 + CODE [0:7]) должна задаваться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

16. 8-битный инкрементирующий счётчик 0 -> N.

Разработать программу восьмибитного счётчика (от 0 до N) с выводом текущего значения в двоичной системе на светодиод. Частота счётчика фиксированная (выбирается студентом). С помощью восьмибитного тумблера должно загружаться максимальное значение счётчика. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка максимального значения счётчика (N) с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

17. Счётчик, инкрементирующий 8-битный с реверсом 0 -> 255 -> 255 -> 0.

Разработать программу восьмибитного счётчика (от 0 до 255 и от 255 до 0) с выводом текущего значения в двоичной системе на светодиоды. Частота счётчика f = f0*(1 + CODE [0:7]) должна задаваться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

18. Гирлянда с эффектом "заполнение".

Разработать программу с последовательно зажигающимися светодиодами, затем гаснущими в той же последовательности. Частота счётчика f = f0*(1 + CODE [0:7]) должна быть настраиваемой. CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

19. Бегущие огни с реверсом.

Разработать программу бегущего светодиода (от 0 светодиода до 7 и от 7 до 0) с настраиваемой частотой f = f0*(1 + CODE [0:7]). CODE [0:7] должен устанавливаться с помощью восьми битного тумблера. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка значения CODE [0:7] с тумблера, при втором нажатии остановка бегущего огня, при третьем

нажатии сброс бегущего огня, при четвёртом нажатии запуск бегущего огня.

20. 8-битный инкрементирующий счётчик 0 -> N.

Разработать программу восьмибитного счётчика (от 0 до N) с выводом текущего значения в двоичной системе на светодиод. Частота счётчика фиксированная (выбирается студентом). С помощью восьмибитного тумблера должно загружаться максимальное значение счётчика. Загрузка значения тумблера в программу должна осуществляться с помощью внешнего прерывания от кнопки. При первом нажатии кнопки должна осуществляться загрузка максимального значения счётчика (N) с тумблера, при втором нажатии остановка счётчика, при третьем нажатии сброс счётчика, при четвёртом нажатии запуск счётчика.

7. Вопросы к лабораторной работе.

1. Что такое struct в языке Си? Как объявить новый тип данных типа структура?
2. Что такое указатель? Как работать с указателем на структуру (запись и чтение значений)?
3. Приведение типов в языке Си?
4. Что такое прерывание? Какие прерывания есть в микроконтроллере STM32F103? Где в программе задаются адреса обработчиков прерываний?
5. Алгоритм работы прерывания? Чем отличается прерывание от вызова функции?
6. Что такое приоритет прерываний и зачем он нужен?
7. Какие отказы могут возникнуть при работе с прерываниями?
8. Что такое push-pull и open-drain? Приведите примеры использования GPIO с настройкой схемы push-pull и open-drain.
9. Поразрядные операции в языке Си. Применение поразрядных операций при установке и сбросе битов в регистрах.
10. Что такое битовый флаг и битовая маска в языке Си? Как ими пользоваться?
11. Модуль RCC. Функциональное назначение. Описание регистров.
12. Модуль GPIO. Функциональное назначение. Описание регистров.
13. Контроллер NVIC. Функциональное назначение. Описание регистров.

8. Литература.

1. **Джозеф Ю. Ядро cortex-m3 компании ARM.**

Глава 7. Исключения.
Глава 8. Контроллер вложенных векторных прерываний и управление прерываниями
Глава 9. Прерывания.
Глава 11. Работа с прерываниями/исключениями

2. **Брайан Керниган и Деннис Ритчи. Язык программирования Си.**

Глава 6. Структуры.

Приложение 1. Инструкции языка ассемблера для ядра Cortex M3

Название	Описание	Операция
Инструкции пересылки данных		
MOV Rd, Rn	Пересылка из одного регистра в другой	Rd ← Rn
MOVW Rd, #const	Запись 16-битного числа в младшие разряды регистра	RdLSB ← const
MOVT Rd, #const	Запись 16- битного числа в старшие разряды регистра	RdMSB ← const
LDR Rd, [Rn, Src]	Чтение 32-разрядного числа из памяти в регистр	Rd31:0 ← mem[Rn + Src]
LDRH Rd, [Rn, Src]	Чтение 16-разрядного числа из памяти в регистр	Rd15:0 ← mem[Rn + Src]
LDRB Rd, [Rn, Src]	Чтение 8-разрядного числа из памяти в регистр	Rd7:0 ← mem[Rn + Src]
STR Rd, [Rn, Src]	Запись 32-разрядного числа из регистра в память	mem[Rn + Src] ← Rd31:0
LDR Rd, =label	Чтение адреса метки	Rd← addres(label)
LDR Rd, =const	Запись 32-битного числа в регистр	Rd ← const
PUSH{Rd}	Запись значения из регистра в стек	mem[SP] ← Rd SP - 4
POP {Rd}	Чтение значения из стека в регистр	Rd ← mem[SP] SP + 4
Примечание: Rd, Rn – регистры общего назначения; Src – регистры общего назначения или константа (#const).		
Арифметические инструкции		
ADD Rd, Rn, Src	Сложение	Rd ← Rn + Src
ADD Rd, Src	Сложение	Rd ← Rd + Src
SUB Rd, Rn, Src	Вычитание	Rd ← Rn - Src
SUB Rd, Src	Вычитание	Rd ← Rd - Src
MUL Rd, Rn, Rm	Умножение	Rd ← Rn x Rm (32-бита LSB)
UMULL Rd, Rn, Rm, Ra	Длинное умножение без знака	{Rd, Ra} ← Rn x Rm (64-бита)
SMULL Rd, Rn, Rm, Ra	Длинное умножение со знаком	{Rd, Ra} ← Rn x Rm (64-бита)
UDIV Rd, Rn, Rm	Деление без знака	Rd ← Rn / Rm
SDIV Rd, Rn, Rm	Деление со знаком	Rd ← Rn / Rm
CMP Rd, Src	Сравнение	Заполнение флагов по результатам Rd - Src

Логические инструкции		
MVN Rd, Rn	Пересылка с инверсией (НЕ)	Rd ← ~Rn
AND Rd, Rn, Src	Логическое И	Rd ← Rn & Src
AND Rd, Src	Логическое И	Rd ← Rd & Src
ORR Rd, Rn, Src	Логическое ИЛИ	Rd ← Rn | Src
ORR Rd, Src	Логическое ИЛИ	Rd ← Rd | Src
ORN Rd, Rn, Src	Логическое ИЛИ-НЕ	Rd ← ~(Rn | Src)
ORN Rd, Src	Логическое ИЛИ-НЕ	Rd ← ~(Rd | Src)
EOR Rd, Rn, Src	Логическое исключающее ИЛИ	Rd ← Rn ^ Src
EOR Rd, Src	Логическое исключающее ИЛИ	Rd ← Rd ^ Src
Инструкции перехода		
B label	Без условный переход	PC ← addres(label)
BEQ label	Переход при выполнении условия «=»	PC ← addres(label)
BNE label	Переход при выполнении условия «!=»	PC ← addres(label)
BLT label	Переход при выполнении условия «<»	PC ← addres(label)
BLE label	Переход при выполнении условия «=<»	PC ← addres(label)
BGT label	Переход при выполнении условия «>»	PC ← addres(label)
BGE label	Переход при выполнении условия «=>»	PC ← addres(label)
BL label	Переход с сохранением адреса текущей инструкции	PC ← addres(label) LR ← PC

Приложение 2. Примеры применений инструкций

Применение арифметический инструкций и присваивания.

Язык высокого уровня (C)	Язык низкого уровня (ассемблер)

int a, b, c;
char mas[10];
a = 2;
b = a;
c = 1000000;
mas[2] = c;
	AREA My_DATA, DATA, READWRITE
	
	mas DCB SPACE 10
	
	AREA RESET, CODE, READONLY
	
	Reset_Handler
	; R0 =a, R1 = b; R3 = c; R4 = mas[0]
	MOVW R0, #0x2
	MOV R1, R0
	MOVW R3, #0x4240; 1000000 = 0xF4240
	MOVT R3, #0xF
	; или можно записать LDR R3, =0xF4240
	LDR R4, =mas
	STRB R3, [R4, #0x2]

int a, b, c, d;
a = 2;
b = 5;
c = 3;
d = a + b - c;
	;R0 = a, R1 = b; R2 = c, R3 = d
	MOVW R0, #0x2
	MOVW R1, #0x5
	MOVW R2, #0x3
	ADD R3, R0, R1
	SUB R3, R3, R2

Применение арифметических инструкций и перехода.

Язык высокого уровня (C)	Язык низкого уровня (ассемблер)

int a, x;
if (a == 10)
{
x = 5;
}
	;R0 = a, R1 = x
	CMP R0, #0xA
	BNE L1
	MOVW R1, #0x5
	L1

int a, x;
if (a == 10)
{
x = 5;
}
else
{
x = 2;
}
	;R0 = a, R1 = x
	CMP R0, #0xA
	BNE L1
	MOVW R1, #0x5
	B L2
	L1
	MOVW R1, #0x2
	L2

switch(a){
case 1: b = 20; break;
case 2: b = 50; break;
	;R0 = a, R1 = b
	CMP R0, #1
	MOVWEQ R1, #20
	BEQ EXIT

case 3: b = 100; break;
default: b = 0;
	CMP R0, #2
	MOVWEQ R1, #50
	BEQ EXIT
	CMP R0, #3
	MOVWEQ R1, #100
	BEQ EXIT
	MOVW R1, #0
	EXIT

Пример цикла с использованием инструкций перехода.

Язык высокого уровня (C)	Язык низкого уровня (ассемблер)

int a = 1;
int x = 0;
while (a != 128)
{
a = a * 2;
x = x + 1;
}
	;R0 = a, R1 = x
	WHILE
	CMP R0, #128
	BEQ DONE
	LSL R0, R0, #1
	ADD R1, R1, #1
	B WHILE
	DONE

int i;
int sum = 0;
for (i = 0; i < 10;i++)
{
sum = sum + 1;
}
	;R0 = i, R1 = sum
	MOVW R1, #0
	MOVW R0, #0
	FOR
	CMP R0, #10
	BGE DONE
	ADD R1, R1, R0
	ADD R0, R0, #1
	B FOR
	DONE