Министерство образования и науки Российской Федерации
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А.В. Пуговкин, А.В. Бойченко, Р.В. Губарева, Е.С. Сорокина, А.М. Мукашев

Методическое пособие по программированию микроконтроллеров.

Учебно - методическое пособие

Томск

2015

Оглавление

Введение....3
1. Описание микроконтроллеров....5
1.1 Краткий обзор микроконтроллеров....5
1.2 Описание микроконтроллера STM32F0....5
1.3 Память....6
1.4 АЦП и ЦАП....7
1.5 Коммуникационные интерфейсы....7
1.6 Питание....8
1.7 Таймеры и система тактирования....8
1.8 DMA память....9
2 Описание среды разработки....11
3 Создание нового проекта....12
4 Лабораторный практикум....17
Лабораторная работа №1....17
Лабораторная работа №2....25
Лабораторная работа №3....29
Лабораторная работа №4....33
Лабораторная работа №5....37
Лабораторная работа №6....42

Введение

Данное методическое пособие предназначено для выполнения лабораторных работ с использованием отладочной платы STM32F0DISCOVERY и среды разработки IAR Embedded Workbench IDE. Оно предназначено для обучения студентов базовым навыкам программирования микроконтроллеров семейства STM. Для написания программ необходимо владеть навыками программирования на языке С.

Первый микроконтроллер появился в 1976 году, американская фирма Intel запатентовала свое изобретение - контроллер i8048. Но данный процессор не получил широкого распространения, тогда в 1980 году той же компанией был выпущен следующий микроконтроллер i8051, который стал прорывом в области микропроцессорной техники, благодаря удобной и более расширенной системе команд, гибкости памяти, удачному набору периферийных устройств и относительной дешевизне. На данный момент существует более 200 модификаций микроконтроллеров, совместимых с i8051. Идея создания микроконтроллеров возникла с необходимостью использования схем управления. Инженерами было испробовано много решений, но самым рентабельным оказалось разместить на одном кристалле не только процессор, но и память с устройствами ввода-вывода. Так для современных микроконтроллеров можно выделить следующие особенности архитектуры и системы команд:

- Гарвардская архитектура – разделены области памяти для хранения команд (программы) и данных.
- Интеграция на одном кристалле всех блоков – процессора, ПЗУ, ОЗУ, устройств ввода/вывода, тактового генератора, контроллера прерываний.
- МК с CISC-архитектурой — с полной системой команд (Complicated Instruction Set Computer);
- МК с RISC-архитектурой — с сокращенной системой команд (Reduced Instruction Set Computer), отличающиеся меньшим временем выполнения команд.

Также микроконтроллеры обычно классифицируют по разрядности обрабатываемых чисел: четырехразрядные, восьмиразрядные, шестнадцатиразрядные и тридцатидвухразрядные. Наиболее распространенные восьмиразрядные, ввиду оптимального сочетания цены и возможностей.

В современном мире микроконтроллеры получили массовое распространение вследствие своей многофункциональности и низкой стоимости. С появлением этих устройств появилась реальная возможность автоматизации практически всех процессов, не только промышленного масштаба, но и на бытовом уровне. МК широко используются в вычислительной технике: в материнских платах, контроллерах дисководов жестких и гибких дисков, CD и DVD, калькуляторах, электронике и разнообразных устройствах

бытовой техники, в которой используется электронные системы управления (стиральных машинах, микроволновых печах, посудомоечных машинах, различных роботах, системах "умный дом", и др.), устройствах промышленной автоматики (от программируемого реле и встраиваемых систем до ПЛК), системах управления станками.

1. Описание микроконтроллеров

1.1 Краткий обзор микроконтроллеров

Микроконтро́ллер (англ. *MicroControllerUnit, MCU*) — микросхема, предназначенная для управления электронными устройствами. Типичный микроконтроллер сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ и (или) ПЗУ. По сути, это однокристальный компьютер, способный выполнять простые задачи.

Неполный список периферии, которая может присутствовать в микроконтроллерах, включает в себя:

- универсальные цифровые порты, которые можно настраивать как на ввод, так и на вывод;
- различные интерфейсы ввода-вывода, такие как UART, I²C, SPI, CAN, USB, IEEE 1394, Ethernet;
- аналого-цифровые и цифро-аналоговые преобразователи;
- компараторы(электронная схема, принимающая на свои входы два аналоговых сигнала и выдающая логическую «1», если сигнал на прямом входе («+») больше, чем на инверсном входе («−»), и логический «0», если сигнал на прямом входе меньше, чем на инверсном входе.);
- широтно-импульсные модуляторы;
- таймеры;
- контроллеры дисплеев и клавиатур;
- радиочастотные приемники и передатчики;
- массивы встроенной флеш-памяти;
- встроенный тактовый генератор;

1.2 Описание микроконтроллера STM32F0

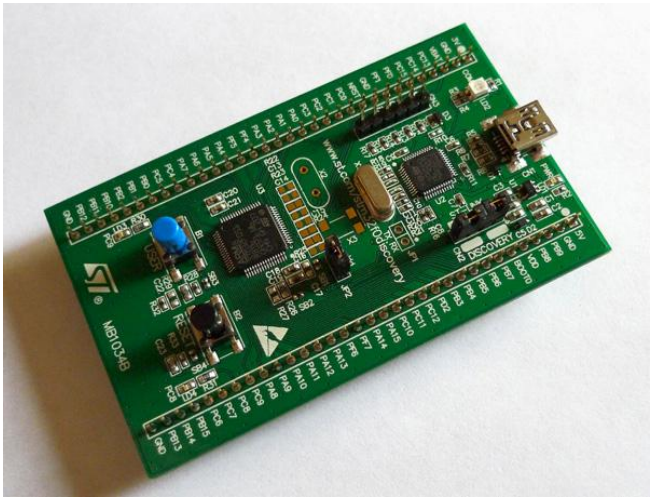

Рисунок 1.1- Внешний вид платы STM32F4

Основные характеристики семейства STM32F0:

- ARM 32-битCortex-M0 CPU;
- Максимальная частота работы - 48МГц;
- До 64кб Flash памяти, до 8кб SRAM памяти;
- DMA-контроллер на 5 каналов;
- 12-бит АЦП на 16 каналов, время конвертирования- 1мкс;
- 12-бит ЦАП;
- До одиннадцати таймеров (16 и 32 разряда);
- Коммуникационныеинтерфейсы: I2C, USART (ISO 7816, LIN, IrDA), SPI, I2S, HDMI;
- Контроллер сенсорных экранов;
- Аппаратное вычисление CRC, 96-битный уникальный ID;
- Часы реального времени (RTC);
- Потребление: 250мкА/МГц в активном режиме, 5мкА в режиме «STOP», 2мкА в режиме «STANDBY» с активным модулем RTC;
- Расширенный температурный диапазон: -40...105°C;

Корпуса UFQFN32, LQFP32, LQFP48, LQFP64.

1.3 Память

Микроконтроллеры STM32F0 имеют до 8 кбайт SRAM-памяти. Она может быть доступна как байт, полуслово (16 бит) или полное слово (32 бита) с нулевым циклом ожидания. Объем встроенной Flash-памяти достигает 64 кбайт.

Для защиты от сбоев, при работе со SRAM-памятью в контроллер встроена проверка на четность. При тяжелых с точки зрения электромагнитных помех условиях работы разработчик может задействовать эту функцию. К 32 битам данных добавляются 4 бита для контроля четности (1 бит на байт). Четность рассчитывается и сохраняется при записи в SRAM. Затем она автоматически проверяется при чтении. Если обратная проверка не проходит, то генерируется ошибка.

Микроконтроллеры STM32F0 были разработаны с учетом работы с операционными системами реального времени. В целом тут больше заслуга компании ARM, которая придерживается данной концепции для всех своих ядер Cortex-M. В обычном 8 или 16-битном микроконтроллере операционная система и приложение делят стек, и сложные программные вложения могут вызвать проблемы с переполнением стека и, соответственно, крах всей системы. Во избежание таких проблем STM32F0 имеет два стека: один для приложений и один - для операционной системы. Это позволяет защитить системную часть кода от фоновых задач операционной системы и, кроме того, уменьшить расход оперативной памяти.

1.4 АЦП и ЦАП

Микроконтроллеры STM32F0 содержат аналогово-цифровой преобразователь ADC (Analog to Digital Converter) с разрешающей способностью 12 бит и скоростью преобразования 1 мкс. Если есть необходимость уменьшить время преобразования, можно уменьшить разрядность результата до 10, 8 или 6 бит. Количество входных аналоговых каналов - 16. Удобная система настроек АЦП позволяет производить однократные и циклические измерения, задавая нужные аналоговые каналы. Для контроля температуры имеется встроенный температурный датчик, подключенный к дополнительному входу АЦП.

Для преобразования цифрового сигнала в аналоговый микроконтроллер содержит цифро-аналоговый преобразователь DAC (Digital to Analog Converter) с разрешающей способностью 12 бит. Очень малая часть 8-битных микроконтроллеров может похвастаться таким набором «аналоговой» периферии с такими хорошими параметрами. Это делает STM32F0 неоспоримым лидером в системах, где требуется быстро и точно производить сбор данных с датчиков и делать их обработку.

1.5 Коммуникационные интерфейсы

Микроконтроллер STM32F0 содержит:

1. Два модуля интерфейса I2C (Inter-Integrated Circuit). Каждый модуль может работать в режиме Master (с поддержкой режима multimaster) или Slave. Интерфейс работает в режиме «Standard» на скорости до 100 кГц, режиме «Fast» на скорости до 400 кГц и «FastPlus» на скорости до 1 МГц.

2. До двух интерфейсов USART (UniversalSynchronous-Asynchronous ReceiverTransmitter), работающих на скоростях до 6 Мбит/с. Также интерфейсы USART имею поддержку интерфейсов LIN, IrDA и смарт-карт. 16

3. Два модуля SPI интерфейса (Serial Peripheral Interface), работающих на скоростях до 18 Мбит/с в режимах Master или Slave. Поддерживается дуплексная и симплексная передача данных.

4. Один модуль I2S интерфейса (Inter-IntegratedSound). Поддерживаются режимы Master или Slave, выполнение операций с 16, 24 и 32-битным разрешением.

5. Один модуль HDMI (High - DefinitionMultimediaInterface) совместимый со спецификацией HDMI-CEC v1.4, работающий в режиме «STOP» для низко потребляющих приложений.

Все коммуникационные интерфейсы поддерживают работу с контроллером DMA, что позволяет STM32F0 поддерживать непрерывную передачу на шинах с максимальными скоростями без остановки или затормаживания работы ядра микроконтроллера. Такой скорости передачи данных невозможно добиться на 8- или 16-

битном микроконтроллере, который не поддерживает DMA. Кроме того, отличительной особенностью STM32F0 по сравнению с 8-битными контроллерами является поддержка HDMI. Это имеет очень важное значение для устройств, нацеленных на потребительский рынок.

1.6 Питание

Для питания микроконтроллера необходим всего один источник питания с напряжением в диапазоне от 2 до 3,6 В. Питание на ядро поступает от встроенного преобразователя напряжения. Для качественной работы аналоговой периферии подача питания на нее осуществляется через отдельные ножки. Для подачи батарейного питания также выделена отдельная ножка. Специально для автономных приложений, требующих низкого потребления, контроллеры STM32F0 имеют три специализированных режима работы.

Режим SLEEP: Потребление уменьшается до нескольких мА. Ядро останавливает свою работу, а вся периферия продолжает работать и пробуждает процессор по наступлению определенного события.

Режим STOP: Потребление падает до нескольких мкА. Все тактирование в зоне питания 1,8 В (от внутреннего стабилизатора) останавливается, данные SRAM и регистров при этом сохраняются. Пробуждение происходит от прерывания модуля EXTI.

Режим STANDBY: Потребление падает до единиц мкА. Внутренний стабилизатор питания 1,8 В отключается, данные SRAM и регистров не сохраняются. Пробуждение происходит от прерывания часов реального времени, общего сброса или возрастающего фронта на ножке WKUP. Часы реального времени могут продолжать свою работу.

1.7 Таймеры и система тактирования

STM32F0 содержит 11 разнообразных таймеров разрядностью 16 или 32 бита: таймер с расширенными функциями (управление двигателями), таймеры общего назначения, два сторожевых таймера (независимый и оконного типа) и 24-битный системный таймер.

Сам факт большей разрядности и большего количества таймеров по сравнению с 8-разрядной архитектурой говорит о существенном преимуществе STM32F0. Кроме этого STM32F0 имеет два сторожевых таймера, одним из которых является оконный сторожевой таймер. Эти два таймера могут работать в режиме пониженного энергопотребления и обеспечивают очень высокий уровень надежности работы приложений, недостижимый в большинстве 8- и 16-битных микроконтроллеров.

Для тактирования микроконтроллера и встроенных часов реального времени потребуются внешние кварцы на частоты 4...32 МГц и 32,768 кГц соответственно. При

критичной конечной цене устройства можно серьезно сэкономить, используя встроенные генераторы. Стоит отметить, что внутренняя система тактирования периферии имеют очень высокую функциональность и гибкость.

Также отметим защитную систему Clock Security System (CSS), которая позволяет микроконтроллеру перейти к тактированию от внутреннего RC-генератора при отказе внешнего. Это обеспечивает работоспособность системы при неисправности кварца, отсутствии контакта при плохой пайке или же сбое внешнего источника тактового сигнала.

1.8 DMA память

Контроллер прямого доступа к памяти DMA (Direct Memory Access) содержит пять каналов. Посредством использования контроллера DMA можно осуществлять автономную передачу данных из памяти в память, из периферийного устройства в память, из памяти в периферийное устройство. При передаче значительных объемов данных модуль DMA может использовать кольцевой буфер. DMA работает со всей наиболее важной периферией: SPI, I2S, I2C, USART, таймеры, АЦП и ЦАП.

Использование контроллера DMA стало неотъемлемой частью приложений, которым необходимо перемещать большие объемы данных, независимо от того, происходит ли вывод информации на внешнее устройство, или же чтение.

В подавляющем большинстве 8-разрядные микроконтроллеры не имеют в своем арсенале таких мощных инструментов, как DMA. В традиционной 8-разрядной архитектуре для перемещения каждого байта данных необходимо задействовать процессор. При перемещении нескольких байтов необходимо несколько циклов процессорного времени на каждый байт, поскольку требуются дополнительные команды для организации циклов.

При использовании DMA в STM32F0 весь блок данных может быть перемещен без привлечения ресурсов ядра микроконтроллера. После того как программа настроит работу контроллера DMA, управляющего перемещением данных, перемещение происходит в фоновом режиме. Фактически процессор переходит в низкопотребляющий режим, так как работа ядра приостанавливается. Это также дает выигрыш в потреблении энергии микроконтроллера и системы в целом.

Наличие DMA-контроллера может также значительно упростить и ускорить процесс разработки. Рассмотрим чтение данных с высокоскоростной шины передачи данных, например, SPI. При использовании 8-битных контроллеров разработчику придется использовать многочисленные прерывания либо постоянно опрашивать флаги готовности, а в промежутке между этими действиями встраивать основной код программы. В STM32F0 процессор может работать независимо от интерфейса, и это позволяет

разработчикам программы использовать его для других задач, не беспокоясь о потере данных.

Поскольку архитектура STM32 использует внутреннюю матрицу шин, то контроллер DMA может быть использован в сочетании с любой внутренней памятью и периферией. Например, контроллер DMA может быть настроен на регулярное считывание и сохранение данных из АЦП в памяти. После завершения очередной операции чтения АЦП отключается до следующего чтения. На самом деле, матрица шин в сочетании с пятиканальным контроллером DMA позволяет STM32 F0 поддерживать выполнение кода из флэш-памяти параллельно с другими операциями чтения и записи данных из памяти в память, периферии в память или памяти в периферию.

Возможности компилятора также могут помочь разработчикам использовать технологии DMA наиболее эффективно. Так IAR Embedded Workbench, например, имеет функцию, которая автоматически изменяет данные программы для максимального использования DMA. Это позволяет разработчикам достичь высокой эффективности без долгой оптимизации расположения данных. Компилятор достигает хорошего результата путем анализа того, как данные используются приложением.

Рассмотрим программу, которая копирует две различные структуры данных с использованием DMA. Каждая операция копирования требует отдельной операции DMA. Тем не менее, после того, как компилятор соотносит структуры данных в памяти, они могут быть скопированы с помощью одной передачи DMA. Заметим, что каждый процессор может использовать DMA по-своему. Например, Keil DK-ARM, используя абстракции, такие как DMA, делает это через API, что предотвращает привязывание кода к конкретному процессору. Это позволяет разработчикам перенести код на другие устройства STM32, зная, что код, использующий DMA, по-прежнему работает оптимально.

2 Описание среды разработки

В нашем лабораторном практикуме используется среда разработки IAREmbeddedWorkbenchforARM (EWARM).

Данная среда поддерживает большое количество микропроцессоров и микроконтроллеров, построенных на базе архитектуры ARM. Весь список микроконтроллеров можно увидеть на их сайте www.iar.com .IARподдерживает различные отладочные интерфейсы, имеет встроенный отладчик, симулятор.

Компания IAR бесплатно предлагает для ознакомления 2 версии продукта. А именно:

- полнофункциональная версия с ограничением использования в 30 дней;
- версия без ограничения использования по времени, но генерирующая код не более 32 Кб.

Для ознакомления с азами программирования микроконтроллеров нас устроит вторая версия.

3 Создание нового проекта

1. Скачать с сайта STMicroelectronix архив «STSW-STM32048», который содержит библиотеку StandartPeripheralsLibrary, примеры её использования и шаблоны проектов для разных сред.

2. Извлечь из архива папки «Utilities» и «Libraries» в папку, где будут находится проекты разработанных прошивок.

3. Создать папку «Empty_Project», в которой будет находиться шаблонный проект. Данная папка должна находиться в одной директории с «Utilities» и «Libraries».

4. Создать в папке «Empty_Project» папки «src» и «inc».

5. В папку «Empty_Project» извлечь из архива папку «EWARM», которая находится по адресу STM32F0xx_StdPeriph_Lib_V1.5.0\Projects\STM32F0xx_StdPeriph_Templates

В результате этих 5 шагов должна получиться следующая иерархия(рисунок 3.1):

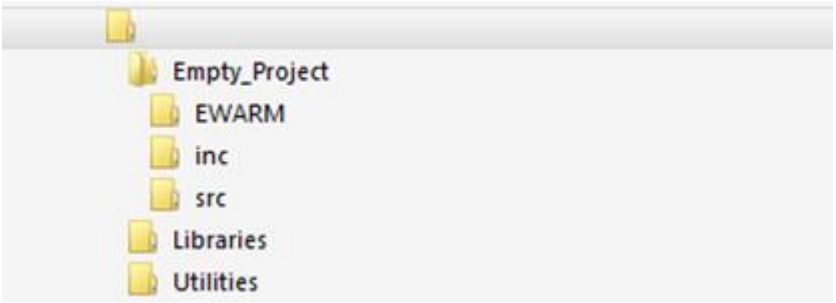

Рисунок 3.1. Иерархия папок

6. В папку «inc» извлечь файлы «main.h», «stm32f0xx_it.h» и «stm32f0xx_conf.h», которые находятся по адресу

STM32F0xx_StdPeriph_Lib_V1.5.0\Projects\STM32F0xx_StdPeriph_Templates.

(main.h – заголовочный файл для main.c

stm32f0xx_it.h – заголовочный файл, содержащий заголовки функций работы с прерываниями

stm32f0xx_conf.h – заголовочный файл, подключающий все файлы библиотеки StdPeriph)

7. В папку «src» извлечь файлы «main.c», «stm32f0xx_it.c» и «system_stm32f0xx.c», которые находятся по адресу

STM32F0xx_StdPeriph_Lib_V1.5.0\Projects\STM32F0xx_StdPeriph_Templates.

(main.c – содержит функцию main

stm32f0xx_it.c – содержит функции работы с прерываниями

system_stm32f0xx.c – файл, отвечающий на настройку режима тактирования)

8. Открываем IAR Embedded Workbench ивыбираем File->Open->Workspace

9. В всплывающем меню заходим в папку «EWARM» и выбираем файл «Project.eww».

10. В **иерархии** проекта (**НЕ** из директории) удаляем из папок все файлы (кроме файлов а папке Output), так как они ссылаются не на те адреса. Получим следующее (рисунок 3.2):

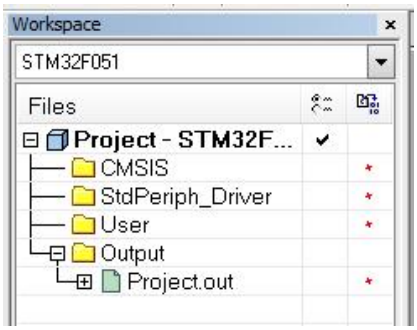

Рисунок 3.2 Иерархия проекта

11. Добавим в папку CMSIS файл «startup_stm32f051.s» по адресу Libraries\CMSIS\Device\ST\STM32F0xx\Source\Templates\iar

(startup_stm32f051.s – файл, содержащий стартовые функции для микроконтроллера, написанные на языке ассемблер)

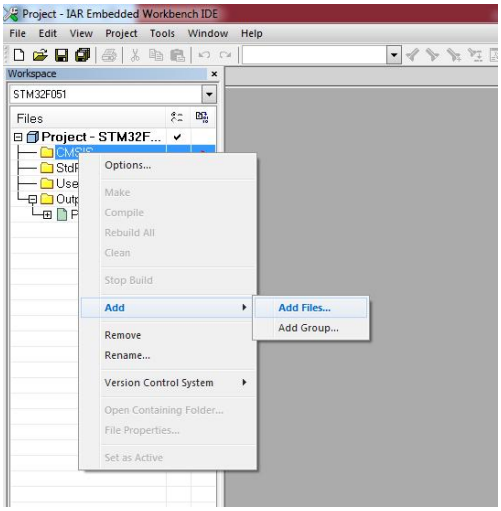

Рисунок 3.3. Добавление файлов

12. Добавим в папку «CMSIS» файл «system_stm32f0xx.c» по адресу Empty_Project\src

13. Добавим в папку «StdPeriph_Driver» файлы «stm32f0xx_rcc.c» и «stm32f0xx_gpio.c», находящиеся по адресу Libraries\STM32F0xx_StdPeriph_Driver\src

(stm32f0xx_rcc.c – отвечает за работу с тактированием периферии,

stm32f0xx_gpio.c – отвечает за работу с портами ввода/вывода)

14. Удалим папку «User» и добавим в иерархию папку «src». В эту папку добавим файлы, находящиеся в соответствующей папке из директории Empty_Project. Получим следующий результат, показанный на рисунке 3.4.

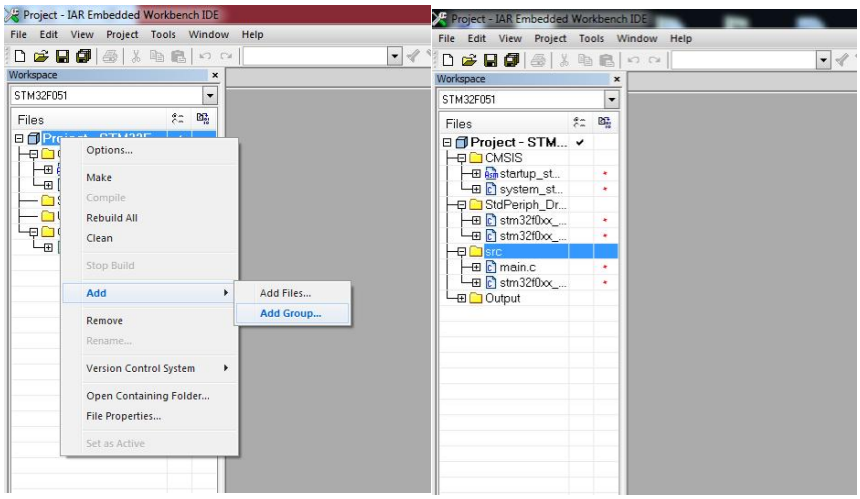

Рисунок 3.4.

15. Откроем свойства проекта (рисунок 3.5)

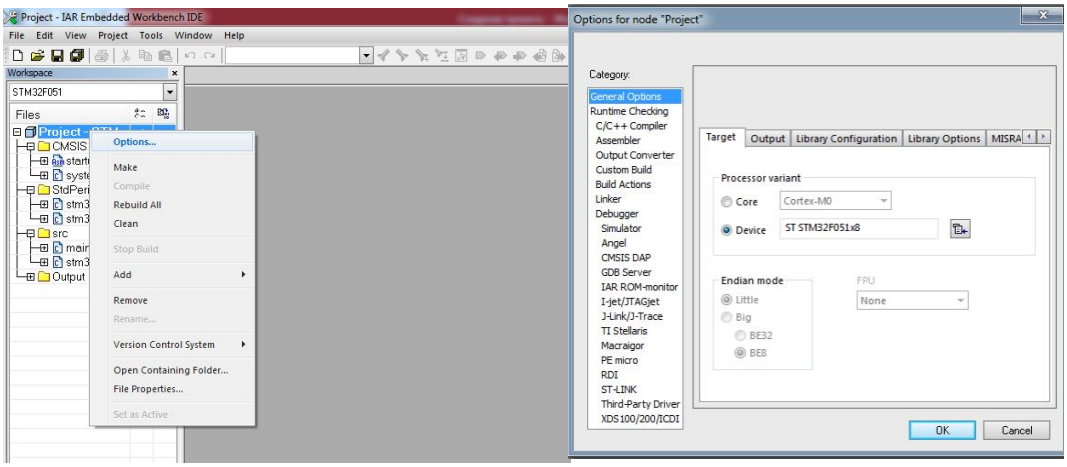

Рисунок 3.5. Свойства проекта

16. Выбираем категорию «C/C++ Compiler» и вкладку Preprocessor(рисунок 3.6.)

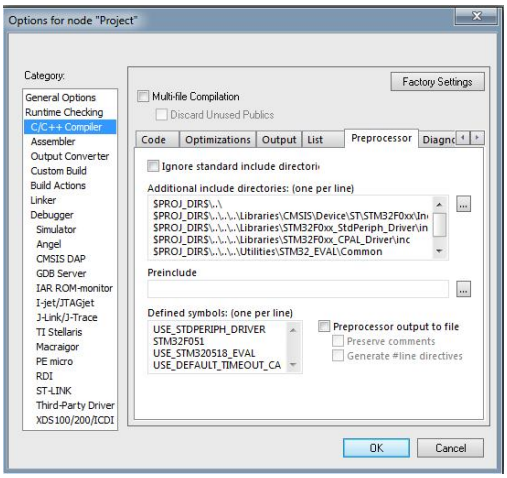

Рисунок 3.6. Выбор категории

17. Исправим адреса библиотек на следующие действующие:

$PROJ_DIR$\..\

$PROJ_DIR$\..\..\Libraries\CMSIS\Device\ST\STM32F0xx\Include

$PROJ_DIR$\..\..\Libraries\STM32F0xx_StdPeriph_Driver\inc

$PROJ_DIR$\..\..\Libraries\STM32F0xx_CPAL_Driver\inc

$PROJ_DIR$\..\..\Utilities\STM32_EVAL\Common

$PROJ_DIR$\..\..\Utilities\STM32_EVAL\STM320518_EVAL

$PROJ_DIR$\..\src

$PROJ_DIR$\..\inc

Фактически на этом процесс создания проекта завершается и проект можно использовать для компиляции прошивок к микроконтроллеру.

Так как процесс создания нового проекта довольно длителен и неизменен каждый раз, создавать таким образом проект не обязательно. Достаточно выполнить эти шаги один раз, а затем копировать созданный проект Empty_Project и переименовывать его под реализуемую задачу.

Необходимо упомянуть ещё один важный пункт:

- Формат и имя файла прошивки

Их можно установить в свойствах проекта во вкладке «OutputConverter». Для программатора ST-Link, который находится на плате, которую мы будем использовать, должен быть установлен тип «Intelextended». В результате прошивка будет компилироваться в формате *.hex.

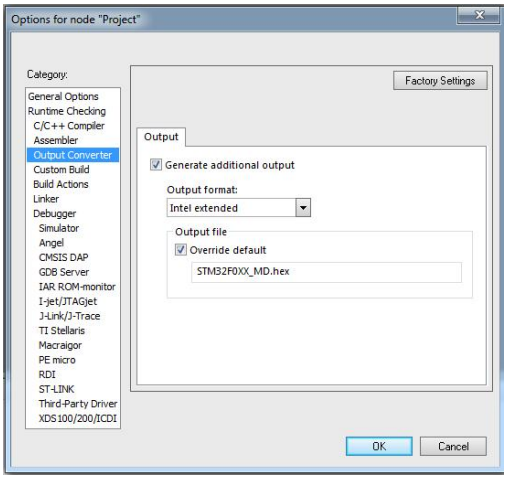

Рисунок 3.7.Вкладка «OutputConverter».

- Выбор типа отладчика

Установить тип отладчика можно во вкладке «Debugger»(рисунок 3.8)

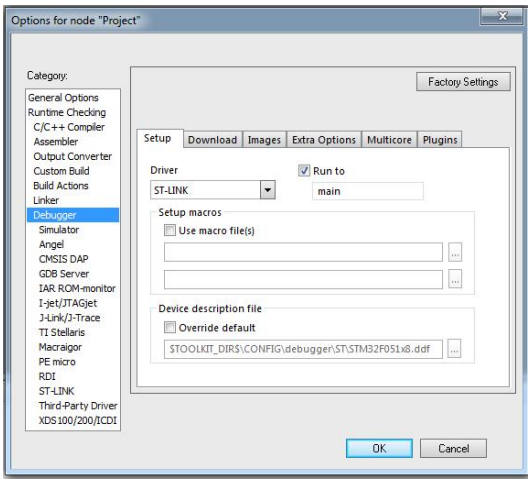

Рисунок 3.8. Установка типа отладчика

4 Лабораторный практикум

Лабораторная работа №1

Повторение языка Си

1. Цель работы: Повторение основ программирования на языке Си. Разработка первых программ.

2. Теоретические сведения

2.1. Компиляция

Си (англ. *C*) — это язык высокого уровня.

Программа, написанная на языке Си, является **текстовым** файлом с расширением.с.Такая программа преобразуется с помощью **компиляции** в программу низкого уровня, состоящую из машинных кодов.

Компиляция состоит из следующих этапов:

1. *Препроцессирование*. Препроцессор добавляет к исходному файлу `*.c` или`*.cpp`заголовочные файлы `*.h, *.hpp` иконстанты. Получается единый текстовый файл.

2. *Трансляция***.** Единый текстовый файл передается *транслятору*. На выходе транслятора возникает файл с ассемблерным текстом.

3. *Ассемблирование.*Ассемблерный текст передаётся программе, которая преобразует его в *объектный файл*`*.obj`.

4. *Линковка*. Линковщик объединяет все поданные ему объектные файлы с библиотечными файлами и формирует один большой файл.

2.2 Заголовочные файлы

Структура сокращенной программы на языке Си может быть такой:

```
#include "stdafx.h" //подключение заголовочных файлов
int main() //заголовок основной программы
{
 //тело основной программы
}
```

Функция `main`– главная функция, с которой начинается выполнение программы. Директива `#include` подключает библиотеки.

2.3 Типы данных

В языке С имеется следующий набор стандартных типов данных:

целые числа (`int, long, unsigned int, unsigned long, bool);`

вещественные числа с плавающей точкой (float, double, longdouble);

указатели;

символьные переменные (char);

тип void.

Диапазоны значений стандартных типов данных приведены в таблице ниже.

Тип	Диапозон значений	Размер(байт)
Bool	true и false	1
Char	-128 .. 127	1
unsigned char	0 .. 255	1
Int	-32 768 .. 32 767	2
unsigned int	0 .. 65 535	2
Long	-2 147 483 648 .. 2 147 483 647	4
unsigned long	0 .. 4 294 967 295	4
float	3.4e-38 .. 3.4e+38	4
double	1.7e-308 .. 1.7e+308	8
long double	3.4e-4932 .. 3.4e+4932	10

2.4 Операторы

Операторы предназначены для осуществления действий и для управления ходом выполнения программ.

Условный оператор:

if(условие) (оператор1).

Более сложная *форма условного оператора* содержит ключевое слово else(иначе):

if (условие) (оператор1); else (альтернативный оператор).

Оператор выполнения цикла с предусловием (пока):

while (условие) [тело цикла].

Оператор выполнения цикла с постусловием (выполняется, пока):

do [тело цикла] while(условие).

Оператор выполнения цикла, содержащий слово for, заголовок и тело цикла:

for (начальные значения; условие; оператор) [тело цикла].

Цикл for используется тогда, когда количество повторений цикла заранее известно илиможет быть вычислено.

Оператор безусловного перехода:

`goto` [метка].

Оператор выхода из цикла и перехода к следующему оператору:

`break.`

Оператор завершения текущего шага цикла и перехода к новому шагу (не выходя из цикла):

`continue.`

Оператор возврата из функции:

`return.`

2.5 Указатели

Указатель — это переменная, содержащая адрес другой переменной. Применение указателя определяется следующими присвоениями:

```
px = &x; //присвоение переменной px адреса переменной x.
y = *px;  //присвоение переменной y величины, адрес которой содержится в переменной px.
```

2.6 Подпрограммы

Подпрограмма (процедура, функция) – это именованная часть программы, к которой можно обращаться из других частей программы.

Функция имеет следующее оформление: имя функции, аргументы функции и тело функции в фигурных скобках. Перед именами функции и именами аргументов ставятся их типы.

Краткий пример функции `vyxod` может быть таким:

```
float vyxod (int n, fload vhod) { тело функции }.
```

2.7 Массивы

Массив - это конечная совокупность данных одного типа. Способы представления массивов поясним конкретными примерами:

```
int x[10];//массив, состоящий из 10 целых чисел (вектор)
```

```
int a[2][5]; //двумерный массив или матрица, состоящая из 2 строк и 5 столбцов
```

В языке Си понятие массива тесно связано с понятием указателя. Имя массива само является указателем на первый элемент массива. Доступ к членам массива можно получать как через имя массива и индекс, так и через указатель на элемент массива и индекс.

2.8 Структура (struct)

Структура - это совокупность данных одинакового или различного типа, обозначенная одним именем. Данные еще называют элементами.

В качестве примера структуры можно взять учетную карточку одного сотрудника предприятия. Элементами такой структуры являются: табельный номер сотрудника, его имя, пол, дата рождения, адрес. Некоторые из этих элементов сами могут оказаться структурами. Например, имя, дата рождения, адрес состоят из несколько частей – элементов другого уровня других структур.

Часто объявляют в начале шаблон структуры, который затем используют для создания структур. Для примера с учетной карточкой шаблон структуры может быть таким:

```
structanketa
{
inttab_nom ; // табельныйномер
charfio[30]; //фамилия, имя, отчество
charpol[3]; //пол
chardata;//датарождения
charadres;
};
```

Ключевое слово `struct` сообщает компилятору об объявлении шаблона структурысименем `anketa`. Для того, чтобы создать структуру, например, с именем `anketa_1`, следует написать `structanketaanketa_1`;

Созданную структуру с именем `anketa_1` называют структурной переменной или просто переменной. Когда объявлена структурная переменная, компилятор выделяет необходимый участок памяти для размещения всех ее элементов. При объявлении структуры можно одновременно объявить одну или несколько переменных, например,

```
struct anketa anketa_1, anketa_2, anketa_3;
```

По другому следом за закрывающей правой фигурной скобкой, заканчивающей список элементов, может следовать список структурных переменных:

```
struct anketa
```

```
{…} anketa_1, anketa_2;
```

Доступ к структурной переменной осуществляется с помощью оператора «точка»:

```
имя_структуры.имя_элемента;
```

2.9. Логические операторы сдвига

Оператор сдвига влево: «<<». Когда оператор сдвига влево (<<) выполняется над некоторым значением, все биты, составляющие это значение, сдвигаются влево. Связанное с этим оператором число показывает количество бит, на которое значение должно переместиться. Биты, которые сдвигаются со старшего разряда, считаются потерянными, а на место младших битов всегда помещаются нули.

Оператор сдвига вправо: «>>». Операция сдвига вправо (>>) сдвигает разряды левого операнда вправо на количество позиций, указываемое правым операндом. Выходящие за правую границу разряды теряются. Для типов данных без знака (unsigned) освобождаемые слева позиции заполняются нулями. Для знаковых типов данных результат зависит от используемой системы. Освобождаемые позиции могут заполняться нулями либо копиями знакового (первого слева) разряда.

3. Структура программы

Так как при программировании микроконтроллера мы будем часто прибегать к структурам и библиотекам, то в качестве примера напишем программу, в которой будем заполнять и выводить на экран элемент типа структура. При этом саму структуру опишем в отдельном файле.

В программе опишем структуру sklad, которая будет хранить данные о хранящихся на складе овощах.

Проект будет состоять из трёх файлов:

Main.c – содержит функцию main().

Sklady.h – содержит описание структуры и заголовки функций для работы с ней.

Sklady.c – содержит описание функций для работы со структурой.

Файл «sklady.h»

Структуры описываются следующим образом:

```
struct sklad
{
  int carrot;
  int potato;
  int tomato;
  char * adress;
```

};

Где sklad – имя структуры. В фигурных скобках перечислены члены структуры. Переменные типа int будут хранить количество овощей на складе. Указатель на char по факту является строкой и будет хранить адрес склада.

Чтобы более удобно выводить хранящиеся в структуре данные, напишем для этого специальную функцию со следующим заголовком:

void printsklad (struct sklad * sample);

В качестве аргумента функция берет указатель на элемент объявленной нами структуры.

Файл «Sklad.c»

В начале этого файла необходимо подключить файлы «sklady.h» и «stdio.h». Последний позволяет использовать функцию printf, которая будет выводить значения переменных в консоль. Для работы вывода данных необходимо в настройках IAR указать, что вывод сообщений от printf передается в окно TerminalIO через библиотеку семихостинга (диалог Options проекта ->GeneralOptions ->LibraryConfiguration ->Librarylow-levelinterfaceimplementation ->stdout/stderr ->Viasemihosting).

В этом файле опишем объявленную ранее функцию printsklad(), которая будет выводить адрес склада и количество килограмм каждого овоща на складе:

```
void printsklad (struct sklad * sample)
{
  printf ("На складе по адресу %s содержится:\n",(*sample).adress);
printf ("Морковки: %d кг\n",(*sample).carrot);
  printf ("Картофеля: %d кг\n",(*sample).potato);
  printf ("Помидоров: %d кг\n",(*sample).tomato);
}
```

Функция printf работает по следующему принципу: текстовые сообщения передаются в кавычках (" "), далее, через запятую, идёт имя переменной, которая будет выводиться в данном сообщении. Сообщение пишется целиком, в место, где должно быть значение переменной вписывается специальный спецификатор (для целых чисел - %d, для символьных строк - %s).

Вывести на экран значения всех полей структуры нельзя, возможно лишь получать значения определённых членов структуры с помощью оператора «точка», например “sample.tomato”, где sample – имя элемента структуры, а tomato – имя переменной-члена структуры.

Так как функция принимает в качестве аргумента адрес структуры, при обращении к ней необходимо использовать оператор “*”, чтобы получить значение по адресу.

Файл «main.c»

В данном файле объявим элемент описанной ранее структуры:

structskladsLen;

Заполним значения полей структуры:

sLen.carrot=200;

sLen.potato=100;

sLen.tomato=50;

sLen.adress="Lenina";

И передадим её адрес в описанную ранее функцию с помощью оператора “&”:

printsklad(&sLen);

4. Ход работы

4.1.Задание на лабораторную работу

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

Файл «sklady.h»:

```
struct sklad
{
  int number;
  int carrot;
  int potato;
  int tomato;
  char * adress;
};

void printsklad(struct sklad * sample);
```

Файл «sklady.c»:

```
#include "sklady.h"
#include "stdio.h"

void printsklad(struct sklad * sample)
{
printf("На складе по адресу %s содержится:\n",(*sample).adress);
printf("Моркови: %d кг\n",(*sample).carrot);
  printf("Картофеля: %d кг\n",(*sample).potato);
  printf("Помидоров: %d кг\n",(*sample).tomato);
}
```

Файл «main.c»:

```
#include "sklady.h"
```

```
int main()
{
  struct sklad sLen;
  sLen.carrot=200;
  sLen.potato=100;
  sLen.tomato=50;
  sLen.adress="Lenina";

  printsklad(&sLen);

  return 0;
}
```

4.3. Индивидуальные задания

1. Перевоз овощей с двух складов на третий.

Дописать к имеющейся программе функцию, которая будет возвращать элемент структуры, содержащий сумму имеющихся овощей на двух складах, передаваемых функции в качестве аргументов.

2. Количество имеющегося картофеля.

В имеющейся программе объявить массив из 5 складов и посчитать количество имеющегося на них картофеля.

3. Сортировка овощей.

Дописать к имеющейся программе функцию, которой будут передаваться адреса 3-х элементов структуры. В результате на первом складе должна быть только сумма моркови с трёх складов, на втором – картофеля, на третьем – помидоров.

Лабораторная работа №2

Порты ввода/вывода(General-purposeinput/output).Библиотека SPL.

1. Цель работы: Изучить основные принципы работы GPIO. Ознакомиться с библиотекой StandartPeriphlibrary(SPL) и ее использованием.

2. Теоретические сведения

Чтобы взаимодействовать с внешним миром – получать и передавать информацию – у микроконтроллера имеются порты ввода/вывода. У микроконтроллера STM32F051R8, используемого в лабораторных работах, 5 портов, содержащих в сумме 55 выводов.

2.1.Режимы работы выводов

Пользовательские выводы могут быть сконфигурированы в один из следующих режимов работы:

- Input floating - это высокоимпедансный вход (Hi-Z), он же плавающий, так как не имеет подтяжки к питанию или земле, поэтому его логическое состояние (1 или 0) всегда определяется напряжением на нём.
- Input pull-up - это вход с подтяжкой к питанию, т.е. между входом и питанием включен подтягивающий резистор (номинала порядка кОм). Подтягивающий резистор позволяет выходу находиться либо в высоком(“1”, подтянут к питанию), либо в низком(“0”, подтянут к земле) состоянии, когда к выходу не приложено внешнее напряжение. Это позволяет избежать спонтанных появлений 0 и 1 на входе.
- Input-pull-down - по аналогии с предыдущим, этот вход — с подтяжкой к земле. При отсутствии напряжения имеет низкое логическое состояние (“0”).
- Analog – это аналоговый вход или выход. Это касается отдельных периферийных блоков(АЦП, ЦАП).
- Outputopen-drain - "выход с открытым стоком".
- Outputpush-pull – “двухтактный выход”. Самый часто используемый тип выхода: подали 0 — выход подключился к земле, 1 — подключился к питанию.
- Alternate function push-pull - уже описанный ранее двухтактный выход, только для альтернативной функции – периферии (SPI, USART…)
- Alternate function open-drain -соответственно выход с открытым стоком, так же для альтернативной функции.

Альтернативные функции, соответствующие выводам, описаны в Datasheet на микроконтроллер (с. 31).

Table 13. Pin definitions (continued)

Pin number					Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
LQFP64	UFBGA64	LQFP48/UFQFPN48	LQFP32	UFQFPN32					Alternate functions	Additional functions
7	E1	7	4	4	NRST	I/O	RST		Device reset input / internal reset output (active low)	
8	E3	-	-	-	PC0	I/O	TTa		EVENTOUT	ADC_IN10
9	E2	-	-	-	PC1	I/O	TTa		EVENTOUT	ADC_IN11
10	F2	-	-	-	PC2	I/O	TTa		EVENTOUT	ADC_IN12
11	A2	-	-	-	PC3	I/O	TTa		EVENTOUT	ADC_IN13
12	F1	8	-	0	VSSA	S			Analog ground	
13	H1	9	5	5	VDDA	S			Analog power supply	
14	G2	10	6	6	PA0	I/O	TTa		USART2_CTS, TIM2_CH1_ETR, COMP1_OUT, TSC_G1_IO1	ADC_IN0, COMP1_INM6, RTC_TAMP2, WKUP1
15	H2	11	7	7	PA1	I/O	TTa		USART2_RTS, TIM2_CH2, TSC_G1_IO2, EVENTOUT	ADC_IN1, COMP1_INP
16	F3	12	8	8	PA2	I/O	TTa		USART2_TX, TIM2_CH3, TIM15_CH1, COMP2_OUT, TSC_G1_IO3	ADC_IN2, COMP2_INM6
17	G3	13	9	9	PA3	I/O	TTa		USART2_RX, TIM2_CH4, TIM15_CH2, TSC_G1_IO4	ADC_IN3, COMP2_INP

Перед использованием вывода он должен быть сконфигурирован в один из вышеперечисленных режимов работы. Это можно сделать несколькими способами:

- Непосредственно через регистры микроконтроллера

- Используя библиотеку StandartPeripheralLibrary (SPL)

В данном лабораторном курсе для настройки и работы с периферией будет использоваться библиотека SPL.

StandartPeripheralLibrary – библиотека, созданная компанией STMicroelectronics. Она содержит функции и структуры для настройки и работы с периферией. Эти функции и структуры берут на себя работу с регистрами, значительно упрощая написание прошивки.

3. Структура программы

В качестве примера использования GPIO напишем программу, в которой с помощью микроконтроллера будет загораться светодиод после нажатия кнопки.

Все функции, классы и константы отвечающие за GPIO представлены в файлах *stm32f0xx_gpio.h* и *stm32f0xx_gpio.c*.

-Инициализация кнопки и светодиода

Нужная кнопка подключена к 4-ому выводу порта C, а светодиод – к 8-ому выводу порта A. (В файле «Распиновка» описано к каким выводам подключены кнопки, светодиоды и прочие детали макета)

За конфигурацию портов отвечает структура *GPIO_InitTypeDef*, поэтому объявим переменную такого типа:

GPIO_InitTypeDefport;

Как и для любой другой периферии сначала необходимо включить тактирование используемых портов (в данном случае Аи С):

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);

Функции для работы с тактированием представлены в файлах *stm32f0xx_rcc.h* и *stm32f0xx_rcc.c*.

Структура *GPIO_InitTypeDef*содержит переменные:

GPIO_Pin - отвечает за номер вывода порта, которую мы хотим настроить,

GPIO_Speed –отвечает за скорость работы порта,

GPIO_Mode - отвечает за режим работы,

GPIO_PuPd – отвечает за подтяжку к земле или питанию.

Для заполнения каждого из этих полей в файлах *stm32f0xx_gpio.h* и *stm32f0xx_gpio.c*.существуют соответствующие константы.

После того как структура GPIO_InitTypeDefзаполнена, чтобы передать её порту, нужно вызвать функцию GPIO_Init(). Функция имеет два аргумета: первый – идентификатор порта, параметры которого необходимо установить, второй – адрес заполненной структуры GPIO_InitTypeDef.

- Включение светодиода по нажатию кнопки

В ходе выполнения программы микроконтроллер должен постоянно опрашивать состояние кнопки (нажата – 1, отпущена – 0) с помощью функции GPIO_ReadInputDataBit(), аргументами которой служат идентификатор порта и идентификатор вывода. Исходя из состояния вывода, подключенного к кнопке, на выводе, подключенном к светодиоду, должна устанавливаться единица либо ноль. Для этого существуют следующие функции:

GPIO_SetBits() –установить на передаваемом в аргументе выводе единицу

GPIO_ResetBits() –установить на передаваемом в аргументе выводе ноль

4. Ход работы

4.1.Задание на лабораторную работу

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

```
voidinitAll(void);

voidinitAll()

{

GPIO_InitTypeDef port;
   void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct); //

   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);
```

```
    GPIO_StructInit(&port);
    port.GPIO_PuPd = GPIO_PuPd_DOWN;
    port.GPIO_Mode = GPIO_Mode_IN;
    port.GPIO_Pin = GPIO_Pin_4;
    port.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &port);
    port.GPIO_Mode = GPIO_Mode_OUT;
    port.GPIO_OType = GPIO_OType_PP;
    port.GPIO_Pin = GPIO_Pin_8;
    port.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOA, &port);
}
int main()
{
    uint8_t buttonState = 0;
    initAll();  //инициализациявыводов
    while(1)
    {
        buttonState = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
         if (buttonState == 1) //проверкасостоянияк нопки
        {
            GPIO_SetBits(GPIOC, GPIO_Pin_8);//включениесветодиода
        }
        else
        {
            GPIO_ResetBits(GPIOC, GPIO_Pin_8);//отключениесветодиода
}
    }
}
```

4.3. Индивидуальные задания

1. Реализовать сумматор двоичных чисел

В ходе работы микроконтроллер должен считывать с переключателей два 4-х значных двоичных числа и выводить их сумму на светодиоды.

2. Реализовать декодер

В ходе работы микроконтроллер должен считывать с переключателей двоичное число – номер светодиода, который должен гореть.

Лабораторная работа №3
Использование таймеров на микроконтроллере STM32F0

1 Цель работы: освоить инициализацию и использование таймеров на микроконтроллере STM.

2 Теоретические сведения

2.1 Таймеры

Микроконтроллер STM32F0 в общей сложности имеет 11 таймеров: 1 16-битный таймер с расширенными возможностями, 5 16-битных и 1 32-битный таймеры общего назначения, 1 базовый таймер, 2 сторожевых таймера и 1 системный таймер. В данной лабораторной работе мы будем использовать только таймеры общего назначения.

Таймеры – один из основных типов периферии микроконтроллеров. Они используются для организации временных задержек, выполнения каких-то периодических событий, генерации ШИМ (широтно-импульсной модуляции) и различных других применений, жестко связанных со временем. О ШИМ будет более подробно рассказано в лабораторной работе №4.

2.2.Прерывания

При работе с различной периферией, в том числе и таймерами, часто используют прерывания. Прерывания – это особый механизм, который позволяет микроконтроллеру быстро реагировать не происходящие события, например, переполнение таймера. В обычных условиях микроконтроллер последовательно выполняет алгоритм, прописанный в функции main(). Но как только происходит событие, для которого включено прерывание, микроконтроллер сразу же устанавливает флаг прерывания и переключает выполнение на функцию обработчик прерывания. По завершении выполнения обработчика микроконтроллер возвращается к выполнению основного алгоритма с того момента, на котором прервался (рисунок 2.1).

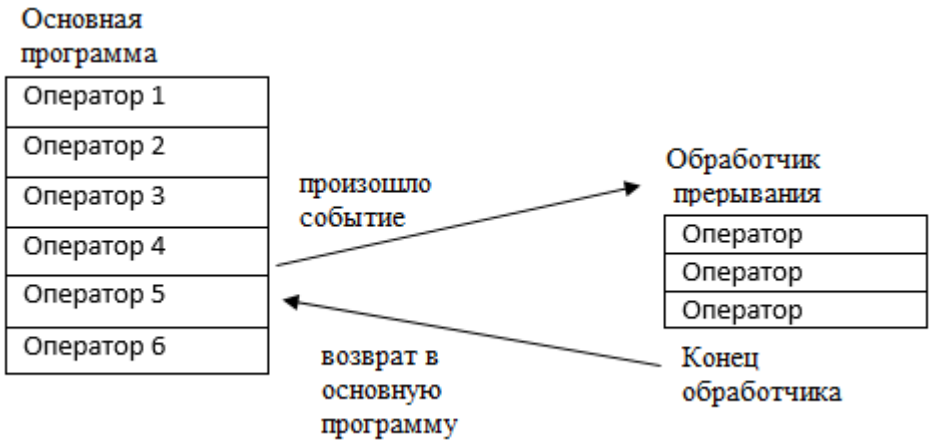

Рисунок 2.1. Ход программы при возникновении прерывания

Чтобы микроконтроллер снова не перешёл в обработчик необходимо очистить флаг прерывания по окончанию его обработки. Список прерываний, их названия и названия функций-обработчиков жестко заданы микроконтроллером.

3.Структура программы

Как было сказано в предыдущей лабораторной работе, любую периферию можно использовать как через библиотеку SPL, так и непосредственно через регистры, но, начиная с этой лабораторной работы, мы будем использовать периферию только с помощью библиотеки SPL.

В качестве примера инициализации и работы с таймерами напишем программу, в которой с помощью микроконтроллера светодиод будет мигать с частотой 1Гц.

Рассмотрим всё по порядку, пропуская уже знакомые по предыдущим лабораторным работам части.

-Инициализация таймера

Все функции, классы и константы для работы с таймерами представлены в файлах stm32f0xx_tim.h и stm32f0xx_tim.c.

Как и для любой другой периферии сначала необходимо включить тактирование:

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

Для задания основных параметров таймера существует структура TIM_TimeBaseInitTypeDef, поэтому объявим переменную такого типа:

TIM_TimeBaseInitTypeDef timer;

Заполним переменную стандартными значениями:

TIM_TimeBaseStructInit(&timer);

Структура TIM_TimeBaseInitTypeDef содержит переменные:

TIM_Prescaler – значение делителя частоты

TIM_Period – период таймера

TIM_CounterMode – режим работы таймера (направление счета и т.д)

Заполнив переменную структуру TIM_TimeBaseInitTypeDef, передадим её микроконтроллеру с помощью специальной функции:

TIM_TimeBaseInit(TIM2, &timer);

И, наконец, запускаем таймер

TIM_Cmd(TIM2, ENABLE);

-Инициализация прерывания, отвечающего за переполнение таймера

Сначала необходимо разрешить использование прерываний для микроконтроллера функцией

__enable_irq();

Разрешаем прерывания от таймера

NVIC_EnableIRQ(TIM2_IRQn);

И включаем прерывание, отвечающее за переполнение таймера

TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

-Смена состояния светодиода в обработчике прерывания

Функция-обработчик прерывания для таймера называется

voidTIM2_IRQHandler()

Функция, убирающая флаг прерывания

TIM_ClearITPendingBit(TIM2, TIM_IT_Update);

4.Ход работы

4.1.Задание на лабораторную работу

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

```
#include "stm32f0xx.h"
GPIO_InitTypeDef port;
TIM_TimeBaseInitTypeDef timer;
uint16_t previousState;

void TIM2_IRQHandler()
{
 if(previousState==0)
 {
 previousState=1;
 GPIO_SetBits(GPIOC, GPIO_Pin_9);
 TIM_TimeBaseInit(TIM2, &timer);
 TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
 }
 else
 {
 previousState=0;
 GPIO_ResetBits(GPIOC, GPIO_Pin_9);
 TIM_TimeBaseInit(TIM2, &timer);
 TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
 }
}

void initAll()
{
 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);

 GPIO_StructInit(&port);
 port.GPIO_Mode=GPIO_Mode_OUT;
 port.GPIO_OType=GPIO_OType_PP;
 port.GPIO_Pin=GPIO_Pin_9|GPIO_Pin_8;
 port.GPIO_Speed=GPIO_Speed_2MHz;
```

```
  GPIO_Init(GPIOC, &port);

  GPIO_SetBits(GPIOC, GPIO_Pin_8);

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

  TIM_TimeBaseStructInit(&timer);
  timer.TIM_Prescaler=4799;
  timer.TIM_Period=10000;
  TIM_TimeBaseInit(TIM2, &timer);
}

 int main()
{
  __enable_irq();
  initAll();

  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  TIM_Cmd(TIM2, ENABLE);
  NVIC_EnableIRQ(TIM2_IRQn);

  while(1)
  {
  }
}

#ifdef  USE_FULL_ASSERT

void assert_failed(uint8_t* file, uint32_t line)
{
  while (1)
  {
  }
}
```

4.3. Индивидуальные задания

1. Реализовать «бегущую строку» на светодиодах

В ходе работы микроконтроллера светодиоды должны загораться по очереди с частотой 1 Гц.

2. Реализовать сигнализацию

В ходе работы микроконтроллера при нажатии кнопки должны включаться 3 светодиода, мигающие с разной частотой.

3. Реализовать секундомер

В ходе работы микроконтроллера светодиоды должны отображать количество прошедших секунд в двоичном коде, при нажатии кнопки должен происходить сброс.

4. Реализовать таймер

В ходе работы микроконтроллера светодиоды должны отображать количество оставшихся секунд в минуте, считая от запуска программы.

По достижении нуля все светодиоды должны мигать с частотой 1 Гц.

5. Светофор

В ходе работы микроконтроллера 3 светодиода должны работать как светофор. Учесть, что желтый свет светофора горит меньше времени.

Лабораторная работа №4
ШИМ (Широтно-импульсная модуляция)

1. Цель работы: Освоить генерацию ШИМ.

2. Теоретические сведения

2.1.ШИМ

Широтно-импульсная модуляция (ШИМ, англ. *pulse-widthmodulation (PWM)*) — управление средним значением напряжения на нагрузке путём изменения скважности импульсов, управляющих ключом. Различают аналоговую ШИМ и цифровую ШИМ, двоичную (двухуровневую) ШИМ и троичную (трёхуровневую) ШИМ.

ШИМ имеет довольно большую практическую пользу. Например, для управления двигателями с возможностью регулировать скорость, для управления яркостью свечения светодиодов, для генерации постоянного напряжения определенного уровня.

С помощью ШИМ реализована регулировка яркости монитора, т.к. при использовании резистивного делителя большая часть энергии преобразуется в тепло.

2.2.Основные принципы формирования ШИМ

ШИМ представляет из себя импульсы, которые с достаточно большой частотой следуют друг за другом. При этом частота следования импульсов подбирается такой, чтобы принимающая сторона считала сигнал постоянным. Уровень напряжения такого «постоянного» сигнала задается за счет ширины импульса. Например, чтобы получить напряжение «постоянного» напряжения в половину амплитуды импульсов ширина импульса должна быть равна половине от расстояния между импульсами (рисунок 2.1)

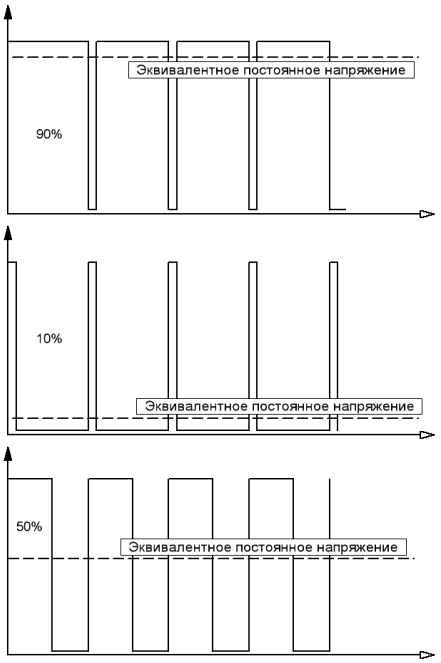

Рисунок 2.1. ШИМ с разной длиной импульса и соответствующее эквивалентное постоянное напряжения

ШИМ в микроконтроллере реализован в периферийных таймерах. ШИМ выводится с помощью настройки соответствующей таймеру ножки.

3.Структура программы

Все функции, классы и константы для работы с таймерами представлены в файлах stm32f0xx_tim.h и stm32f0xx_tim.c.

В качестве примера мы реализуем программу, в которой с помощью ШИМ светодиод плавно загорается и плавно потухает.

Как и в прошлый раз рассмотрим всё по порядку, пропуская уже знакомые по предыдущим лабораторным работам части.

- Инициализация ножки, к которой подключен светодиод.

Ножка, соответствующая таймеру должна быть настроена в альтернативный режим. Настройка альтернативного режима выполняется функцией:

GPIO_PinAFConfig(GPIOC, GPIO_PinSource8, GPIO_AF_1);

В качестве аргументов функция принимает имя порта, имя ножки и константу, отвечающую за выбор альтернативной функции. Константы и соответствующая им периферия указаны в файле stm32f0xx_gpio.h.

- Настройка ШИМ

Для этого объявляем переменную структуры TIM_OCInitTypeDef:

TIM_OCInitTypeDef timerPWM;

Эта структура содержит переменные:

TIM_Pulse – ширина импульса

TIM_OCMode – режим ШИМ-сигнала (выравнивание по границе/по центру)

TIM_OutputState – включение/выключение вывода ШИМ на ножку порта

Затем заполненную переменную структуры необходимо передать функции:

TIM_OC3Init(TIM3, &timerPWM);

и выполнить предзагрузку ШИМ:

TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);

-Смена скважности импульсов в обработчике прерывания

Смену скважности будем производить, меняя значение ширины импульса в структуре и заново передавая структуру в инициализирующую функцию.

4. Ход работы

4.1.Задание на лабораторную работу

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

```
#include "stm32f0xx.h"
GPIO_InitTypeDef port;
TIM_TimeBaseInitTypeDef timer;
TIM_OCInitTypeDef timerPWM;
uint16_t skvaz;
int div;
void TIM3_IRQHandler()
{
  skvaz+=div;
  timerPWM.TIM_Pulse = skvaz;
  TIM_OC3Init(TIM3, &timerPWM);
  if(skvaz>=2000) div=-10;
  if(skvaz==00) div=10;
  TIM_ClearITPendingBit(TIM3, TIM_IT_Update);
}
void initAll()
{
  skvaz=00;
  div=10;
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);
  GPIO_StructInit(&port);
  port.GPIO_Mode=GPIO_Mode_AF;
  port.GPIO_OType=GPIO_OType_PP;
  port.GPIO_Pin=GPIO_Pin_8;
  port.GPIO_Speed=GPIO_Speed_50MHz;
  GPIO_Init(GPIOC, &port);
  GPIO_PinAFConfig(GPIOC, GPIO_PinSource8, GPIO_AF_1);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
  TIM_TimeBaseStructInit(&timer);
  timer.TIM_Prescaler=47;
  timer.TIM_Period=10000;
  TIM_TimeBaseInit(TIM3, &timer);
  TIM_OCStructInit(&timerPWM);
```

```
  timerPWM.TIM_Pulse = skvaz;
  timerPWM.TIM_OCMode = TIM_OCMode_PWM1;
  timerPWM.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OC3Init(TIM3, &timerPWM);
  TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);
}
 int main()
{
  __enable_irq();
  initAll();
  TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);
  TIM_Cmd(TIM3, ENABLE);
  NVIC_EnableIRQ(TIM3_IRQn);

  while(1)
  {
  }
}
#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t* file, uint32_t line)
{
  while (1)
  {
  }
}
```

4.3. Индивидуальные задания

1. Реализовать аналог «гирлянды»

В ходе работы микроконтроллера должны плавно загораться и потухать попеременно чётные и нечётные светодиоды. Использовать 8 светодиодов.

2. Реализовать систему плавной регулировки освещения

В ходе работы микроконтроллера при удержании одной кнопки яркость свечения должна увеличиваться, при удержании другой кнопки яркость свечения должна уменьшаться.

3. Реализовать управление RGB-светодиодом

В ходе работы микроконтроллера при нажатии кнопки светодиод должен последовательно переключаться между цветами радуги.

Лабораторная работа №5
SPI-интерфейс (Последовательный периферийный интерфейс)

1. Цель работы: Освоить инициализацию и использование SPI-интерфейса

2. Теоретические сведения
2.1.Общие сведение

SPI (англ. Serial Peripheral Interface — последовательный периферийный интерфейс) — последовательный синхронный стандарт передачи данных в режиме полного дуплекса, предназначенный для обеспечения простого и недорогого сопряжения микроконтроллеров и периферии.

Устройства, связывающиеся через SPI-интерфейс делятся на Ведущие (Master) и Ведомые(Slave), при этом Ведущим может быть только одно устройство. Только ведущий может инициировать передачу данных.

В SPI-интерфейсе используется 4 провода(рис.2.1):

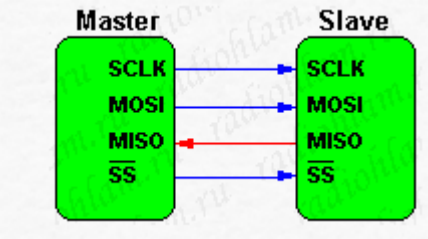

Рисунок 2.1 - SPI-интерфейс

- SCLK– для передачи тактовых импульсов
- MOSI (MasterOut, SlaveIn) – для передачи данных от Ведущего к Ведомому
- MISO (MasterIn, SlaveOut) – для передачи данных от Ведомого к Ведущему
- NSS – для выбора Ведомого

В соединении только одно устройство может быть Ведущим, в то время как Ведомых устройств может быть много. При этом для всех Ведомых устройств используются одни и те же выводы SCLK, MOSI и MISO, а выводов NSSy Ведущего должно быть равным количеству Ведомых устройств (рисунок 2.2).

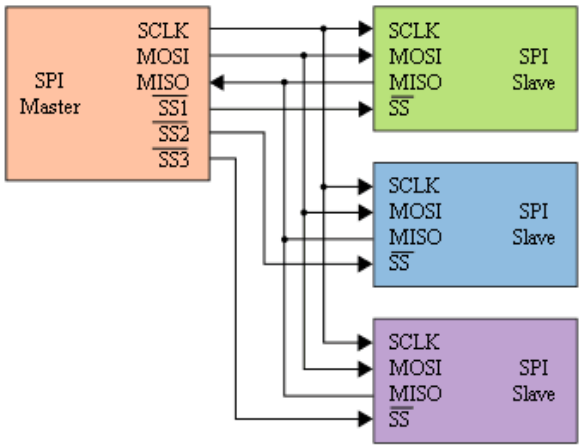

Рисунок 2.2 – Соединение с несколькими Ведомыми

Первоначально на всех NSS-выводах Ведущего должна быть выставлена логическая единица. Чтобы инициировать обмен данных с определённым Ведомым, Ведущий выставляет на соответствующий этому ведомому NSS-вывод состояние логического нуля. Ведомые с высоким уровнем на NSS-выводе в обмене данными при этом не участвуют.

2.2.Основные настройки SPI-интерфейса

У SPI-интерфейса также имеется две основные настройки: CPOL (полярность тактовых импульсов) и CPHA (фаза тактового сигнала). При CPOL=0 начальное состояние CLS – логический ноль, а при CPOL=1 – логическая единица. При CPHA=0 данные считываются по переднему фронту тактового импульса, а записываются – по заднему. При CPHA=1, наоборот, данные считываются по заднему фронту тактового импульса, а записываются – по переднему (рисунок 2.3).

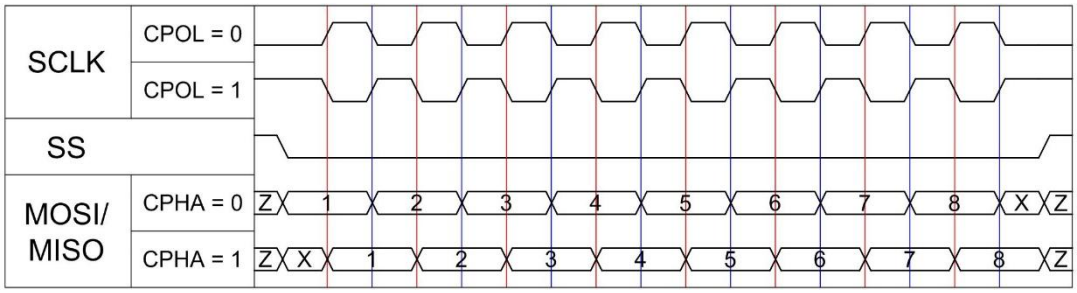

Рисунок 2.3 – Режимы SPI

Количество битов в одном кадре (между падением и подъемом уровня на NSS-выводе) протоколом не закреплено, но обычно используют 8-битный кадр.

У микроконтроллера STM32F051 имеется 2 периферийных SPI-интерфейса. Увеличить количество SPI можно за счёт реализации программным образом (написать в виде кода), так как механизм работы этого интерфейса достаточно прост.

3.Структура программы

Для использования в программе SPI необходимо добавить к проекту файлы библиотеки SPL: stm32f0xx_spi.c и stm32f0xx_spi.h. Они содержат структуры и функции для инициализации и работы с SPI.

В качестве примера реализуем два программы: SPI в режиме Ведущего и SPI в режиме Ведомого.

Мы будем использовать SPI1. Согласно Даташиту выводы распределены следующим образом:

NSS – PA4

SCK – PA5

MISO – PA6

MOSI – PA7

Для инициализации SPI используется структура SPI_InitTypeDef. Эта структура состоит из следующих переменных:

SPI_Direction – отвечает за направление работы SPI (только приём, только передача или полный дуплекс)

SPI_Mode – режим SPI (Master-Ведущий или Slave-Ведомый)

SPI_DataSize – размер кадра (количество бит)

SPI_CPOL – Полярность тактового импульса

SPI_CPHA – Фаза тактового импульса

SPI_NSS – Режим управления NSS-выводом (программно или автоматически)

SPI_BaudRatePrescaler – предделитель частоты тактовых импульсов

SPI_FirstBit – передача старшим/младшим битом вперед

SPI_CRCPolynomial – константа, отвечающая за генерацию контрольной суммы

Затем заполненную структуру передаем через функцию

SPI_Init(SPI1,&spi)

Включается SPI функцией SPI_Cmd(SPI1,ENABLE);

Отправка определённого байта подключенному Ведомому

Отправка данных осуществляется через функцию SPI_SendData8(SPI1,sendData);

4. Ход работы

4.1.Задание на лабораторную работу.

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

```
#include "stm32f0xx_gpio.h"
#include "stm32f0xx_misc.h"
#include "stm32f0xx_rcc.h"
#include "stm32f0xx_spi.h"

GPIO_InitTypeDefport;
SPI_InitTypeDefspi;
uint8_tsendData;
uint16_tcurrent;

voidinitAll()
```

```
{

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA,ENABLE);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC,ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1,ENABLE);


port.GPIO_Mode=GPIO_Mode_AF;
port.GPIO_OType=GPIO_OType_PP;
port.GPIO_Pin=GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
port.GPIO_Speed=GPIO_Speed_50MHz;
GPIO_Init(GPIOA,&port);

port.GPIO_Mode=GPIO_Mode_IN;
port.GPIO_OType=GPIO_OType_PP;
port.GPIO_PuPd=GPIO_PuPd_DOWN;
port.GPIO_Pin=GPIO_Pin_0;
port.GPIO_Speed=GPIO_Speed_50MHz;
GPIO_Init(GPIOA,&port);

port.GPIO_Mode=GPIO_Mode_OUT;
port.GPIO_OType=GPIO_OType_PP;
port.GPIO_PuPd=GPIO_PuPd_NOPULL;
port.GPIO_Pin=GPIO_Pin_9|GPIO_Pin_8;
port.GPIO_Speed=GPIO_Speed_50MHz;
GPIO_Init(GPIOC,&port);

SPI_StructInit(&spi);
spi.SPI_Direction=SPI_Direction_2Lines_FullDuplex;
spi.SPI_Mode=SPI_Mode_Master;
spi.SPI_DataSize=SPI_DataSize_8b;
spi.SPI_CPOL=SPI_CPOL_Low;
spi.SPI_CPHA=SPI_CPHA_2Edge;
spi.SPI_NSS=SPI_NSS_Soft;
spi.SPI_BaudRatePrescaler=SPI_BaudRatePrescaler_4;
spi.SPI_FirstBit=SPI_FirstBit_MSB;
spi.SPI_CRCPolynomial=7;
SPI_Init(SPI1,&spi);
}

intmain()
{
__enable_irq();
initAll();
GPIO_SetBits(GPIOC,GPIO_Pin_8);
SPI_Cmd(SPI1,ENABLE);
sendData=0xCA;
current=0;
while(1)
{
sendData=0xCA;
SPI_SendData8(SPI1,sendData);
GPIO_SetBits(GPIOC,GPIO_Pin_9);
sendData=0x00;
SPI_SendData8(SPI1,sendData);
GPIO_ResetBits(GPIOC,GPIO_Pin_9);
}
}

#ifdefUSE_FULL_ASSERT

voidassert_failed(uint8_t*file,uint32_tline)
{
```

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

В случае использования SPI в режиме Ведомого, инициализация ножек и самого SPI производится аналогично, с отличием в переменной SPI_Mode.

Приём данных осуществляется на основе прерывания от SPISPI1_IRQHandler(). Считывание данных производится функцией SPI_ReceiveData(SPI1) которую можно вызывать при подъеме флага SPI_IT_RXNE, который отвечает за наличие данных на входе.

4.3. Индивидуальные задания:

Следующие индивидуальные задания выполняются попарно (одна группа программирует Ведущий микроконтроллер, другая – Ведомый)

1. Реализовать передачу короткого сообщения через SPI

В ходе работы Ведущего микроконтроллера при нажатии кнопки с определённой временной задержкой должны отправляться коды ASCII на Ведомый. Ведомый должен по приёму байта выводить его на светодиоды.

2. Реализовать дистанционное управление макетами через SPI

В ходе работы на триггерах набираются определённые комбинации. При изменении комбинации триггеров Ведущего микроконтроллеры должны обмениваться текущими состояниями триггеров и выводить полученные комбинации на светодиоды.

Лабораторная работа №6
Универсальный приемопередатчик (USART)

1.Цель работы: Научиться использовать универсальный синхронный/асинхронный приемопередатчик, организовывать передачу данных другим устройствам

2. Теоретические сведения

Универсальный синхронный/асинхронный приёмопередатчик (УАПП, англ. *Universal Asynchronous Receiver-Transmitter, UART*) — узел вычислительных устройств, предназначенный для организации связи с другими цифровыми устройствами. Преобразует передаваемые данные в последовательный вид так, чтобы было возможно передать их по цифровой линии другому аналогичному устройству. Метод преобразования хорошо стандартизован и широко применялся в компьютерной технике.

Представляет собой логическую схему, с одной стороны подключённую к шине вычислительного устройства, а с другой имеющую два или более выводов для внешнего соединения.

3.Структура программы

В качестве примера мы реализуем программу, в которой микроконтроллер опрашивает кнопку, при нажатие которой контроллер вышлет в USART сообщение «Pressed», а если кнопка будет не нажата, то «Not Pressed».

Для использования в программе USARTнеобходимо добавить к проекту файлы библиотеки SPL: stm32f0xx_ usart.c и stm32f0xx_ usart.h. Они содержат структуры и функции для инициализации и работы с USART.

За конфигурацию портов отвечает структура USART *_InitTypeDef*, поэтому объявим переменную такого типа:

```
USART_InitTypeDef usart;
```

Как и для любой другой периферии сначала необходимо включить тактирование:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);
```

4. Ход работы

4.1.Задание на лабораторную работу.

1. Ознакомиться с теоретическими сведениями.

2.Написать, отладить и запустить программу.

3. Выполнить индивидуальное задание.

4.2.Текст программы

```
#include "stm32f0xx_rcc.h"
#include "stm32f0xx_gpio.h"
#include "stm32f0xx_tim.h"
#include "stm32f0xx_usart.h"
#define BAUDRATE 9600
GPIO_InitTypeDef port;
USART_InitTypeDef usart;
uint8_t usartData[10];
uint16_tbutton;
uint16_t usartCounter = 0;
uint16_t numOfBytes;
 void initAll()
{
      RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
   RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE);

   GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_1);
   GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_1);

   GPIO_StructInit(&port);
   port.GPIO_Mode = GPIO_Mode_AF;
   port.GPIO_OType = GPIO_OType_PP;
   port.GPIO_PuPd = GPIO_PuPd_UP;
   port.GPIO_Pin = GPIO_Pin_9;
   port.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_Init(GPIOA, &port);

   port.GPIO_Mode = GPIO_Mode_AF;
   port.GPIO_OType = GPIO_OType_PP;
   port.GPIO_PuPd = GPIO_PuPd_UP;
   port.GPIO_Pin = GPIO_Pin_10;
   port.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &port);
USART_StructInit(&usart);
```

```
    usart.USART_BaudRate = BAUDRATE;
    usart.USART_Parity = USART_Parity_No;
    usart.USART_StopBits = USART_StopBits_1;
    usart.USART_WordLength = USART_WordLength_8b;
    usart.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
    USART_Init(USART1, &usart);

    port.GPIO_Mode = GPIO_Mode_IN;
    port.GPIO_PuPd = GPIO_PuPd_DOWN;
    port.GPIO_Pin = GPIO_Pin_0;
    port.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOA, &port);

    port.GPIO_Mode = GPIO_Mode_OUT;
    port.GPIO_OType = GPIO_OType_PP;
    port.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_8;
    port.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &port);

    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
USART_Cmd(USART1, ENABLE);
}
void setData()
{
    button = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
    if (button == 0)
    {
        usartData = “NotPressed”;
        numOfBytes = 10;
    }
    else
    {
        usartData = “Pressed”;
numOfBytes = 7;
    }
```

```
    usartCounter = 0;
}
 int main()
{
    __enable_irq ();
    initAll();
    NVIC_EnableIRQ(USART1_IRQn);
    while(1)
    {
      if(counter==10000)
      {
        GPIO_SetBits(GPIOC, GPIO_Pin_9);
      }
      if(counter==20000)
      {
        GPIO_ResetBits(GPIOC, GPIO_Pin_9);
        counter=0;
      }
while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
    }
}
void USART1_IRQHandler()
{
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
      uint16_t usartData;
      usartData = USART_ReceiveData(USART1);
      if(usartData=='B')
        GPIO_ResetBits(GPIOC, GPIO_Pin_9);
      else
        GPIO_SetBits(GPIOC, GPIO_Pin_9);
    }
    GPIO_SetBits(GPIOC, GPIO_Pin_8);
    USART_ClearITPendingBit(USART1, USART_IT_RXNE);
}
```