Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

ЦИФРОВЫЕ УСТРОЙСТВА И МИКРОПРОЦЕССОРЫ
ЧАСТЬ 1: МИКРОКОНТРОЛЛЕРЫ

Учебное пособие

Санкт-Петербург

2020

УДК 004.31
Р32

А в т о р ы:
Р. В. Веринский, П. С. Тетерин, Д. К. Фадеев

Цифровые устройства и микропроцессоры : учеб. пособие / Р. В. Веринский [и др.]. – СПб., 2020. – 196 с.

Учебное пособие соответствует образовательному стандарту высшего образования Федерального государственного автономного образовательного учреждения высшего образования «Санкт-Петербургский политехнический университет Петра Великого» по направлению подготовки бакалавров 11.03.01 «Радиотехника», по дисциплине «Цифровые устройства и микропроцессоры».

Представлены сведения о структуре микроконтроллера с ARM архитектурой на примере микроконтроллера STM32F407VG. Разобраны принципы разработки программ для микроконтроллеров и алгоритмы решения типичных задач встраиваемых систем. Рассматриваются принципы взаимодействия с блоками, входящими в состав микроконтроллера.

Предназначено для студентов, аспирантов, преподавателей технических факультетов вузов.

© Санкт-Петербургский политехнический университет Петра Великого, 2020

Оглавление

Введение ... 10
Часть 1: Общие положения ... 12
Получение информации о цифровых устройствах ... 12
Структура микроконтроллера ... 12
Порты общего назначения ... 15
Принцип работы порта ввода-вывода stm32f ... 16
Прерывания ... 19
Таймеры-счётчики ... 21
Таймеры общего назначения ... 21
Системный таймер ... 24
Кнопка ... 25
Схема подключения ... 25
Дребезг контактов ... 26
Алгоритм борьбы с дребезгом ... 27
USART ... 29
Общие положения ... 29
Контроль чётности ... 30
Область применения. ... 31
Протоколы передачи данных. ... 31
Приём данных. ... 32
Буферы приёма. ... 33
DMA ... 35
SPI ... 36
Обзор выводов SPI ... 36
Протокол передачи ... 37
Применение SPI ... 40
Жидкокристаллические индикаторы ... 42
Применение дисплеев ... 42
Дисплей WH1602 D–NGG–CT ... 42

Описание контроллера дисплея 43
АЦП 46
Подключение АЦП 46
Виды АЦП. 46
Датчик температуры LM35 47
Пересчёт значений АЦП в напряжения 48
Цифро-Аналоговый преобразователь 49
Акселерометр 50
Принцип работы акселерометра. 50
Акселерометр LIS302DL 51
Акселерометр LIS3DSH 51
Инкрементальный энкодер 53
Принцип работы 53
Выводы энкодера pec16-4220f-s0024 54
Клавиатура 55
Память 57
Общие положения. 57
Структура Flash 57
Генератор случайных чисел. 60
Общие понятия 60
Центральная предельная теорема 62
Часть 2: Программирование микроконтроллера stm32f407vg 63
Основные библиотеки для микроконтроллеров stm32fxxx 63
Библиотека CMSIS 63
Библиотека StdPeriph 63
Создание проекта в keil uVission 65
Подключение библиотеки CMSIS 67
Настройка библиотеки CMSIS 68
Подключение библиотеки StdPeriph 69
Настройка тактирования микроконтроллера 73

Определение тактовой частоты ядра 74
Программирование платы stm32f4-discovery 75
Использование основных функций компилятора и отладчика. 77
Компиляция проекта 77
Программирование микроконтроллера 77
Поиск определений функций и переменных 77
Использование точек останова 78
Выполнение программы по шагам 78
Использование окна Watch 79
Борьба с ошибками компилятора 80
Определение частоты тактирования ядра МК 83
Программирование портов ввода-вывода. 84
Настройка портов ввода-вывода с помощью библиотеки CMSIS. 84
Пример настройки портов ввода-вывода на библиотеке CMSIS 89
Настройка портов ввода-вывода с помощью библиотеки StdPeriph 90
Пример настройки портов ввода-вывода с помощью библиотеки StdPeriph 94
Программирование таймера 95
Настройка таймера общего назначения 95
Пример инициализации таймера общего назначения 97
Инициализация системного таймера 98
Пример настройки системного таймера 99
Алгоритм реализации длительных временных задержек 99
Как узнать частоту тактирования таймера 101
Алгоритмы борьбы с дребезгом 103
Общий обзор алгоритмов борьбы с дребезгом 103
Алгоритм борьбы с дребезгом с помощью задержки 103
Гистерезисный алгоритм борьбы с дребезгом 106
Настройка USART с помощью библиотеки StdPeriph. 108
Описание общего алгоритма настройки 108
Пример настройки USART 113

Протоколы 114
Формат кодирования данных 114
Примеры наиболее часто используемых полей пакетов 115
Поле команд 115
Поле данных 118
Поле длины 120
Поле контроля целостности 121
Поле стоп байта 121
Поле адреса устройства 123
Разработка протокола передачи данных 123
DMA 125
Структура DMA 125
Пример настройки DMA 132
SPI 135
Общая настройка SPI 135
Пример подключения микросхемы по SPI 136
Реализация протокола микросхемы акселерометра 141
ЖКИ 144
Подключение дисплея к контроллеру 144
Восьми битное подключение. 144
4 битная шина 145
Контроллер HD44780 147
Инициализация контроллера. 150
8-ми битный интерфейс передачи данных 150
4-х битная шина 151
Программирование дисплея 152
Переопределение выводов 152
Функции передачи команды и данных. 154
Инициализация контроллера. 155
Запись строки данных 156

АЦП ... 157
Настройка регулярного канала. ... 157
Пример настройки АЦП ... 161
Программирование ЦАП ... 164
Работа с flash памятью ... 166
Чтение из Flash ... 166
Разблокирование Flash ... 166
Стирание Flash ... 166
Запись в Flash ... 168
Пример чтения и записи ... 168
Работа с памятью ... 170
Программирование энкодера ... 171
Программирование кнопки ... 171
Программирование вращения энкодера ... 171
Принципы программирования клавиатуры ... 175
Настройка генератора случайных чисел ... 179
Подключение библиотеки для работы с SDIO с поддержкой fatfs ... 180
Часть 3: Полезные возможности языка Си ... 184
Использование функции sprintf ... 184
Использование функции atoi ... 184
Директива define ... 185
Создание нового файла ... 186
Часть 4: Алгоритмы ... 189
Алгоритм борьбы с дребезгом ... 189
Алгоритм организации задержки на Systick ... 191
Алгоритм TimeOut. ... 194
Отправка строки с текстом по USART ... 197
Конечный автомат ... 198
Часть 5: Лабораторные ... 205
Как пользоваться таблицами вариантов ... 205

Содержание отчётов 205
Лабораторная 1: Использование библиотеки CMSIS 206
Задача 206
Лабораторная 2: Использование библиотеки StdPeriph 209
Задача 209
Порядок выполнения работы 209
Содержание отчёта. 209
Лабораторная 3: Таймеры 210
Цель 210
Задачи 210
Содержание отчёта: 210
Лабораторная 4: Порты общего назначения в режиме входа 212
Задачи 212
Содержание отчёта: 212
Лабораторная 5: USART 215
Задание 215
Возможные ошибки и их решения 221
Содержание отчёта: 222
Лабораторная 6: LCD 223
Задачи 223
Содержание отчёта: 223
Лабораторная 7: ADC 226
Задачи 226
Содержание отчёта: 226
Лабораторная 8: Память 229
Задачи 229
Содержание отчёта: 229
Лабораторная 9: SPI 230
Задачи 230
Содержание отчёта: 231

Лабораторная 10: Энкодер 232
Задачи 232
Содержание отчёта: 232
Лабораторная 11: ЦАП 233
Задачи 233
Содержание отчёта: 233
Лабораторная 12: Клавиатура 234
Задачи 234
Содержание отчёта: 234
Лабораторная 13: Генератор случайных чисел 235
Задачи 235
Содержание отчёта: 235
Лабораторная 14: Динамическая индикация 236
Задачи 236
Содержание отчёта: 236
Лабораторная 15: SD карта 237
Задачи 237
Содержание отчёта: 237

Введение

Цифровые системы отличаются от аналоговых тем, что работают с сигналами с ограниченным числом состояний. Это уменьшает чувствительность этих систем к шумам. Если шумы меньше дискретности уровней цифрового сигнала, то цировые устройства могут полностью убрать этот шум без потерь. В то время, как аналоговый сигнал так просто избавить от шумов не получается.

Самые простые цифровые комопненты строятся на обычных транзисторах. Например, так можно построить обычный инвертор сигнала, который на вход получает логическую единицу, а на выход выдаёт логический ноль.

Более сложные схемы фировых устройств собираются в отдельное устройство под названием микросхема, которое выполняет некую функцию. Например, это могут быть микросхемы, выполняющие сложение, логические операции и т.д. На наборе таких микросхем можно построить уже довольно сложную схему. Но для новых задач требуется строить и паять новую схему, что не удобно. Особенно, если задача регулярно меняется, что приводит к постоянным переделываниям схемы.

Для решения этого вопроса был разработан отдельный тип цифровых микросхем под названием микроконтроллер. Эта микросхема включает в себя множество цировых устройств, таких, как память, устройтсва выполняющие математические операции, устройства передачи данных и т.д.

Для управления работой микроконтроллера пишется программа, которая определяет задачи, выполняемые микроконтроллером. Теперь, если задача поменяется, то достаточно переписать программу для микроконтроллера.

В результате большую часть задач берёт на себя именно микроконтроллер, а остальная схема максимально проста и универсальна. Поэтому в курсе основной упор делается именно на программирование микроконтроллеров на примере микроконтроллера stm32f407vg.

Часть 1: Общие положения

Получение информации о цифровых устройствах

Микросхемы чаще всего представляют собой некое устройство, выполняющее заданную функцию. На вход подаются некие данные, а микросхема на выход выдаёт ответ. Или генерирует некий сигнал независимо от входных данных.

Для хранения настроек работы микросхемы в ней предусмотрены ячейки памяти, в которых хранятся данные, определяющие режим работы микросхемы. Эти ячейки называются регистры. Запись данных в регистр и чтение из них происходит по определённому интерфейсу. Иногда можно выбрать один из нескольких интерфейсов. Данные же на микросхему передаются либо по тому же интерфейсу, либо по другому. Например, настройка микросхемы кодера производится по последовательному интерфейсу, а данные передаются по параллельному.

Чтобы работать с цифровыми микросхемами необходимо узнать как общие принципы работы этой микросхемы и её возможности, так и конкретное назначение каждого регистра настроек. Чтобы узнать всё это, нужно скачать техническую документацию на эту микросхему с официального сайта производителя этой микросхемы. Из неё можно узнать назначение и адреса регистров микросхемы, напряжение питания, расположение и назначение выводов и многое другое.

Структура микроконтроллера

Микроконтроллер представляет собой законченное устройство, включающее в себя блок обработки поступающей информации и блоки, реализующие различные функции. Среди которых: таймеры, интерфейсы передачи данных, цифровые порты общего назначения. Примерная блок-схема показана на рис. 1.1.

Рис. 1.1

Главным модулем микроконтроллера является арифметико-логическое устройство. Оно выполняет все команды, которые поступают на микроконтроллер. И от его структуры зависит какие команды и как быстро может выполнять микроконтроллер. АЛУ может быть построен на основе различных ядер и иметь разную производительность при одинаковой частоте тактирования. Лучше заранее выяснить какое ядро используется в выбираемом микроконтроллере.

Исполняемую АЛУ программу надо где-то хранить и для этого используется энергонезависимая память ПЗУ - постоянное запоминающее устройство. В неё записывается программа в виде бинарных команд и из неё АЛУ считывает нужные инструкции.

Также АЛУ требуется где-то хранить временные данные (переменные). Для этого предназначена более быстрая, чем ПЗУ энергозависимая память (ОЗУ – оперативное запоминающее устройство). Здесь хранятся данные, потеря которых при выключении питания не критична.

Каждый микроконтроллер имеет множество выводов на своём корпусе. И каждый вывод для чего-то используется. Всего выводы можно поделить на две большие группы: служебные и программируемые. Причём в некоторых

микроконтроллера программируемые выводы могут также использоваться как служебные.

Среди служебных выводов основными являются выводы на которые подаётся напряжение питания микроконтроллера (VDD и земля) и вывод сброса, который заставляет микроконтроллер начать выполнение программы с начала.

Для тактирования микроконтроолера у него есть встроенный генератор, построенный на RC цепочке. Но его стабильность низкая и не позволяет создавать программы с точной привязкой ко времени. Для получения стабильного тактового сигнала есть 2 выводаXTAL. К ним пожет быть подключен либо генератор тактовой частоты, либо цварцевый осциллятор. Во втором случае внутренный генератор микроконтроллера генерирует сигнал, опираясь уже не на RC цепочку, а на этот осциллятор.

Частота кварцевого осциллятора, подключаемого к микроконтроллеру обычно ограничена значениями порядка 20-30 МГц. При этом ядро может работать на частоте выше 100 МГц. Для преобразования входной частоты в частоту ядра используется PLL. Это умножитель-делитель частоты, настраивая параметры которого, можно управлять частотой тактирования ядра и других периферийных устройств, т.к. PLL может отдельно от ядра тактировать другие устройства, такие как таймеры, интерфейсы передачи данных и т.д. Точную функциональность встроенного PLL можно получить из документации на микроконтроллер.

Для организации привязки программы ко времени используются таймеры, который с помощью подсчётов испульсов тактовой частоты позволяют отмерять промежутки времени.

Для общения с внешним миром микроконтроллер использует программируемые выводы, на которые можно установить либо логический ноль, либо логическую единицу, либо подключить один из внутренних модулей, чтобы уже тот выдавал или получал данные.

Конечно этим списком число модулей не ограничивается и каждый микроконтроллер имеет ещё множество различных модулей. Но об этом надо читать техническую документацию на конкретный микроконтроллер.

Для получения технической документации на имеющийся в распорядении микроконтроллер, надо зайти на сайт производителя этого микроконтроллера. Сайт можно найти по названию написанному на корпусе микросхемы. На сайте надо найти страничку этого микроконтроллера и скачать файл reference manual.pdf или с любым другим подобным названием.

Часто назначение выводов и описание корпуса выносят в отдельную документацию. В этом случае требуется скачать production specification.pdf или с аналогичным названием.

Теперь перейдём непосредственно с разбору периферии микроконтроллера.

Порты общего назначения

Общее описание порта ввода-вывода

Порт ввода-вывода является программно-управляемым элементом микроконтроллера. Порт состоит из отдельных выводов, являющихся ножками микроконтроллера. В stm32f порты включают в себя 16 выводов. Такие порты по числу выводов называются 16-ти разрядными. Обычно названия портов обозначаются латинскими буквами A, B, C и т.д. Выводы портов нумеруются цифрами 0, 1, 2 и т.д. Причём нулём нумеруется самый младший бит и далее по старшинству.

Выводы микроконтроллера могут работать в двух режимах: на вход и на выход. В первом случае с ножки снимается текущее напряжение и это значение можно использовать в программе. Например, это может быть сигнал готовности какого-то устройства. Т.е. мы ждём, когда на определённой ножке установится 1. Когда это произошло, мы можем работать дальше, т.к. наш гипотетический прибор готов к взаимодействию с МК. Во втором случае программист задаёт будет ли на выводе логический

ноль или логическая единица. Этот режим может использоваться для управления внешними устройствами. Например, чтобы включить какой-то прибор требуется установить логическую единицу на определённом выводе микроконтроллера.

Под логическим нулём подразумевается напряжение в 0 вольт. Уровень логического нуля для микроконтроллера stm32f4 равен 3.3 В. Напряжения между эти напряжениями могут быть восприняты как равными нулю, так и единице. Порог лежит где-то в районе 1.5 вольт.

Выводы при работе на вход или выход воспринимают исключительно бинарную информацию и могут определять или выдавать только логический ноль или логическую 1.

Отметим, что все выводы микроконтроллера могут работать независимо друг от друга. И один порт может одновременно управлять несколькими устройствами, присоединёнными к разным выводам этого порта. Т.е. один и тот же порт может одновременно установить логическую единицу на один из своих выводов и затем ждать прихода единицу, означающей готовность устройства, на другой вывод.

Также к выводам общего назначения могут подключаться внутренние устройства МК в случае, если им надо выводить наружу какие-то сигналы. При этом порт общего назначения может быть запрограммирован для использования в режиме альтернативной функции. В этом режиме вывод перестаёт работать как вывод общего назначения и подключается к заданному внутреннему устройству МК, которое через этот порт сможет передавать данные.

Последний режим работы выводов в stm32f4 – это работа с аналоговыми сигналами. В этом режиме вывод работает с блоками ЦАП и АЦП микроконтроллера.

Принцип работы порта ввода-вывода stm32f

Блок-схема порта ввода вывода микроконтроллера семейства stm32f представлена на рис 1.2

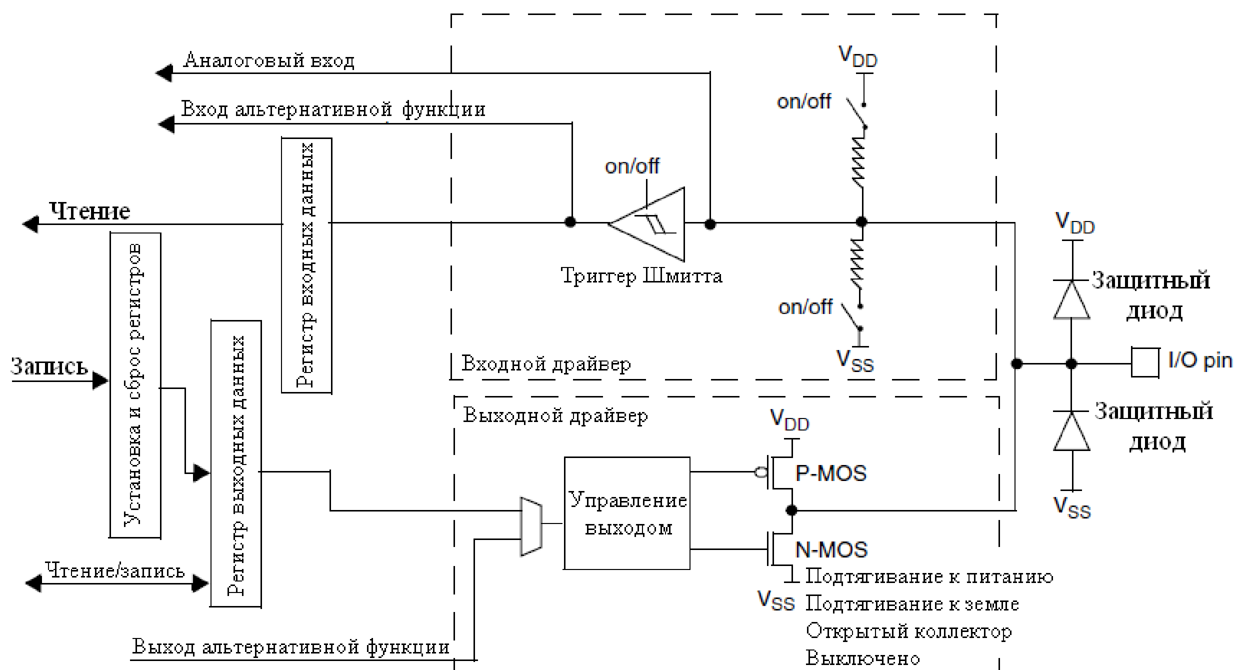

Рис. 1.2

Порт ввода-вывода микроконтроллера работает либо на вход, либо и на выход. Одновременно работать в обоих направлениях выводы микроконтроллера не могут. Поэтому всю структуру порта ввода-вывода можно разделить на два больших блока: блок управления входом и блок управления выходом.

Общими для входа и выхода являются два защитных диода, ограничивающих напряжение, которое может быть подано на вывод микроконтроллера. На принцип работы порта при нормальных условиях работы (напряжение на выводе не превышает напряжения питания МК) эти диоды никак не влияют.

В блоке приёмной части два резистора отвечают за подтяжку вывода к питанию или к земле. Это может пригодиться, если предполагается, что входной сигнал может оказаться «висящим в воздухе», т.е. быть никуда не подключенным. Кроме того, вход можно никуда не подтягивать.

Далее идёт разделение в зависимости от настроек. Сигнал может пойти на альтернативную функцию (например, USART) или на регистр входных значений. В данном регистре записывается состояние порта при работе его на вход. Узнать о состоянии порта можно, причитав значение из этого регистра.

В передающей части в регистр даннных записывается число, которое надо подать на выводы порта. Это число поступает на пару транзисторов, которые отвечают за уровень нуля и единицы. Здесь определяется режим работы на выход. Если включена работа обоих транзисторов, тогда логический ноль выставит на выходе 0 вольт, а логическая единица выставит на выход напряжение питания (обычно 3.3 вольта). Во втором случае работает только транзистор, подключенный к нулю. Этот режим называется «открытый коллектор». В этом режиме логический ноль тоже устанавливает 0 вольт на выходе, но логическая единица оставляет вывод не подключенным. И для правильной работы его требуется потянуть к питанию резистором.

Второй режим работы вывода на выход может использоваться для совместимости с пятивольтовой микросхемой. В этом случае вывод ставится как выход с открытым коллектором и подтягивается к пяти вольтам.

При настройке вывода как альтернативная функция, внутренний блок сам устанавливает напряжения на выводе. Значения в регистре данных порта при этом игнорируются.

Отметим, что в микроконтроллере есть так называемые битовые маски, которые позволяют записывать в регистр данных побитно. Но это сильно повлияло на скорость работы порта при работае на выход.

Прерывания

Возьмём простую задача: требуется вскипятить воду в чайнике. Для этого наполняем чайник, ставим на плиту, а дальше надо дождаться пока он закипит. И тут можно поступить по-разному. Во-первых можно постоянно смотреть как он закипает и выключить сразу, как только закипит. Во-вторых, можно уйти и периодически возвращаться и выключить чайник, когда обнаружим его кипящим. И есть третий способ, когда чайник известит вас о закипании овды звуковым сигналом. Тогда вы бросаете то, чем занимались до этого и идёте выключать чайник.

Разберём плюсы и минусы каждого из подходов. Плюсы и минусы первого варианта очевидны. Чайник выключается сразу, как только закипит. Это плюс. Но минус в том, что вы просто так тратите время на ожидание закипания воды.

Второй способ позволяет выполнять другие задачи, но чайник будет выключен через неопределённое время после закипания. А может быть вообще не выключен, если вы полностью займётесь другим делом и не будете проверять чайник.

При использовании третьего способа, можно выполнять любые действия не думая о проверке чайника и выключить максимально быстро после закипания. Но эти плюсы достигаются путём некоторого усложнения устройства.

Закипание чайника – это некое событие. Оно может быть любым. В микроконтроллере тоже есть события. Например, программа ожидает прихода байта данных по некоему интерфейсу. Тут точно также можно постоянно или периодически проверять флаг, показывающий, что байт пришёл. А можно настроить аналог свистка, который укажет программе, что байт пришёл. Такой «свисток» в микроконтроллерах называется «модуль прерываний».

Как работает этот модуль? Микроконтроллер может реагировать на множество разных событий от разных модулей. Чтобы он не разнрывался между постоянно возникающими событиями, реакции на все эти события по-умолчанию выключена. Чтобы микроконтроллер отреагировал на некое событи, сначала требуется его разрешить в настройках. Чаще всего флагов разрешения два: Один разрешает глобальные прерывания вообще или целого модуля, а второй разрешает уже события каждого модуля. В результате происходит некое событие, микроконтроллер проверяет разрешены ли прерывания по этому событию. Если они разрешены, то вызывается функция, привязанная к этому событию.

В результате достаточно настроить модуль, разрешить прерывания от этого модуля и прерывания от конкретного события и описать функцию, название которой строго задано и обычно заканчивается Handler. Теперь, если программа попала в эту функцию, то можно быть уверенным, что нужное событие свершилось и требуется его обработать.

Например, некий системный счётчик считает от 0 до 255. В момент, когда он переходит из 255 в 0, происходит событие «переполнение счётчика». Если разрешить реакцию на это событие, то будет в этот момент аппаратно будет вызвана функция SysTick_Handler. И в этой функции должно быть описано то, что будет выполнено при переполнении этого счётчика.

Каждому событию микроконтроллера сопостояваляется адрес, в который перейдёт программа при возникновении этого события. Этот адрес носит название «вектор прерывания». Большинство прерываний построены по векторному принципу с разпределением по приоритетам. Но есть ещё один тип прерываний – это радиальные прерывания, подключенные отдельно от векторых. Эти прерывания имеют самый высокий приоритет и выполняются быстрее векторных. Обычно таким прерыванием является «сброс».

Как же обрабатываются прерывания? Как уже было сказано, при появлении события, проверяется флаг разрешения, если он установлен, то

выполнение текущей программы останавливается и выполнение программы переносится в соответствующую функцию. Эта функция выполняется и затем выполнение программы возвращается к тому месту, на котором было прервано выполнение основной программы.

Отметим, что обычно в микроконтроллерах выполнение функцияии прерывания не может прерыватся для выполнения другой функции прерывания. Но этот момент требуется уточнять в конкретной документации на заданный микроконтроллер. У прерываний есть приоритеты и у некоторых микроконтроллеров прерывания с более высоким приоритетом таки могут прерывать выполнение функций прерываний с более низким приоритетом. Но обычно прерывания просто ставятся в очередь путём выставления флага, говорящего о том, что событие произошло. Но т.к. флаг не может быть выставлен дважды, два одинаковых события встать в очередь не смогут и второе будет пропущено.

Из этого следует, что программа в функции прерывания должна выполняться максимально быстро. В функции прерывания должно находиться только то, что надо сделать немедленно по событию. Идеально, чтобы в этой функции просто выставлялся программный флаг, что функция вызывалась, а полноценную обработку уже совершать в основном цикле.

Таймеры-счётчики

Таймеры общего назначения

Основная задача таймеров-счётчиков – это подсчёт числа импульсов, поступающих к ним на вход. При появлении восходящего фронта (переход из 0 в 1) на входе таймера-счётчика, тот увеличивает (или уменьшает в зависимости от режима работы) значение на 1 в регистре, отвечающем за хранение числа подсчитанных импульсов.

Рассмотрим блок-схему таймера-счётчика более подробно (рис 1.3).

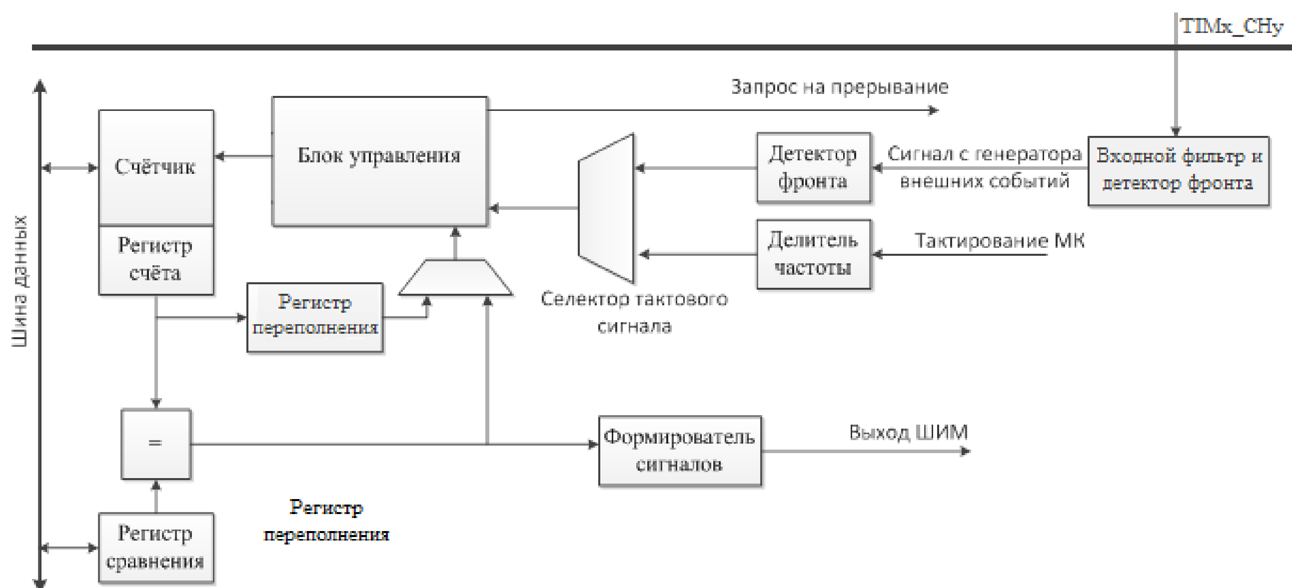

Рис 1.3

Основой любого таймера-счётчика является блок “Счётчик”, включающий в себя “регистр счёта”, который может быть 8-ми, 16-ти или 32-х разрядным. Его значение может быть либо увеличено, либо уменьшено на единицу “Блоком управления”. Исходя из этого, различают 3 режима работы:

А) инкрементный. В этом режиме значение регистра таймера увеличивается на единицу каждый принятый такт до достижения максимального (или заданного пользователем) значения. После чего значение збрасывается в 0 и цикл начинается заново.

Б) декрементный. В этом режиме значение регистра таймера уменьшается на единицу каждый принятый такт пока не достигнет нуля. После чего значение опять устанавливается равным максимальному и цикл начинается заново.

В) инкрементно-декрементный. В этом режиме таймер с начала увеличивает значение своего регистра на единицу, а по достижении максимального (или заданного пользователем) значения уменьшает его на единицу за принятый такт до тех пор, пока не достигнет нуля. После чего цикл начинается заново.

Решение об увеличении или об уменьшении значения “Счётчика” “Блок управления” принимает при получении восходящего фронта на входе от

“Селектора тактового сигнала”. “Селектор тактового сигнала” отвечает за выбор источника импульсов. В качестве входной импульсной последовательности может быть, как внутренний сигнал тактирования микроконтроллера, так и внешний сигнал, поступающий на определённый вывод микроконтроллера.

При использовании внутреннего тактирования таймера, сигнал на “Блок управления” подаётся через делитель частоты, что позволяет настроить требуемую частоту прихода импульсов и, как следствие, скорость счёта “Счётчика”. В этом режиме работы, благодаря высокой стабильности частоты следования импульсов, таймер-счётчик может работать как таймер, отсчитывая заданные промежутки времени. Это получается из-за того, что период следования импульсов фиксирован. Поэтому можно воспользоваться формулой:

$t=T*N$

Где T – период следования импульсов, а N – число импульсов, пришедших за отслеживаемый период времени. Т.е., если при тактовой частоте таймера в 1 кГц на него пришло 10 импульсов, значит прошло 1/1000*10=10мс. Разумеется, точность такого подсчёта времени определяется периодом следования импульсов.

При подсчёте импульсов, приходящих с внешнего входа таймер-счётчик может работать в нескольких режимах.

А) Таймер. Если период входного сигнала постоянен.

Б) Счётчик. Когда требуется посчитать приходящие по линии импульсы.

В) Снятия данных с энкодера. Некоторые счётчики могут обрабатывать данные с энкодера в автоматическом режиме. Для этого требуется подключить выводы a и b энкодера к двум выводам (каналам 1 и 2) таймера и настроить этот таймер на работу с энкодером. В этом случае “Счётчик” будет уменьшаться или увеличиваться в соответствии с вращением ручки энкодера будет. Более подробно про энкодер можно прочитать в соответствующем разделе.

Счётчик в инкрементном режиме работы сбрасывается в ноль по достижении значения, записанного в блок “Регистр переполнения”. При включении таймера обычно там устанавливается максимально возможное число. Например для 16-тибитного таймера это 0xFFFF. При переходе из максимального значения в ноль таймер генерирует запрос на прерывание, если оно разрешено.

Кроме прерываний по переполнению, таймер может генерировать прерывания по сравнению текущего значения “Счётчика” с содержимым регистра “=”. На основе этих двух событий (переполнение и сравнение) таймер может генерировать ШИМ сигнал.

Системный таймер

Зачастую возникает необходимость строгой привязки времени выполнения программы к реальному времени. Например, в случае, когда требуется сгенерировать сигнал нужной длительности. При этом настройка обычного таймера для решения подобной задачи слишком громоздка. В итоге для решения подобных задач в микроконтроллере предусмотрен специальный таймер реального времени. Включение этого таймера, вызывает возникновение прерываний через строго определённые промежутки времени, задаваемые самим разработчиком. В функцию, вызываемую этим прерыванием, вставляется код, который должен выполняться через заданные интервалы времени. Например, это может быть запуск аналогово-цифрового преобразования, т.к. зачастую необходимо, что бы частота дискретизации сигнала была стабильной. Остальная же часть функционирования таймера скрыта. Это позволяет не отвлекаться на настройку таймера и больше времени отвести для решения непосредственно поставленной задачи.

Кроме того, на системном таймере могут отсчитываться заданные промежутки времени любой длительности. Подробнее об этом будет рассказано во второй части методического пособия.

Кнопка

Схема подключения

Кнопка обычно может находиться в одном из двух состояний: нажата и отжата. При этом в зависимости от подключения на ней могут быть разные потенциалы. Например, если кнопка подключена, так как изображено на рис 1.3, на отжатой кнопке в точке Input будет высокий потенциал, а при нажатой - низкий. Это связано с тем, что при отжатой кнопке точка Input подключена через резистор к питанию. Если кнопка нажата, то всё напряжение падает на резисторе, а точка Input оказывается закорочена на землю.

Если же кнопка подключена, как показано на рис 1.4, то, наоборот, при отжатом состоянии на кнопке будет ноль, а при нажатом, напряжение питания (логическая единица).

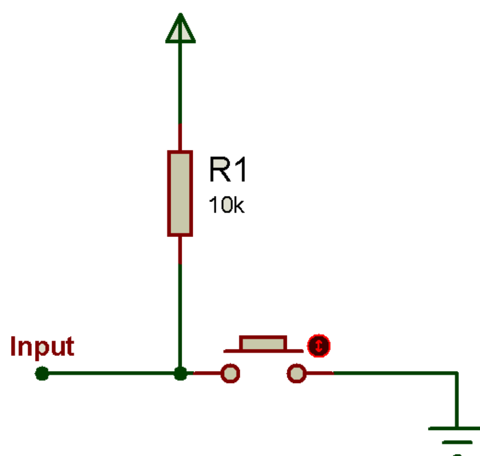

Рис 1.3

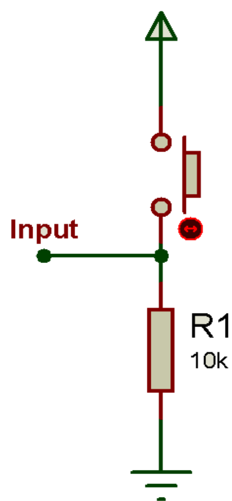

Рис 1.4

Дребезг контактов

При моделировании кнопка считается идеальной и её модель выглядит, как показано на рис 1.5.

Рис 1.5

Фронты при нажатии считаются идеальными. В реальности обычно при изменении состояния кнопки появляется эффект дребезга контактов. Этот эффект заключается в том, что пластины соприкоснувшись друг с другом (при нажатии) не остаются в таком состоянии, а какое-то время болтаются и сигнал на кнопке принимает вид, как показано на рис. 1.6.

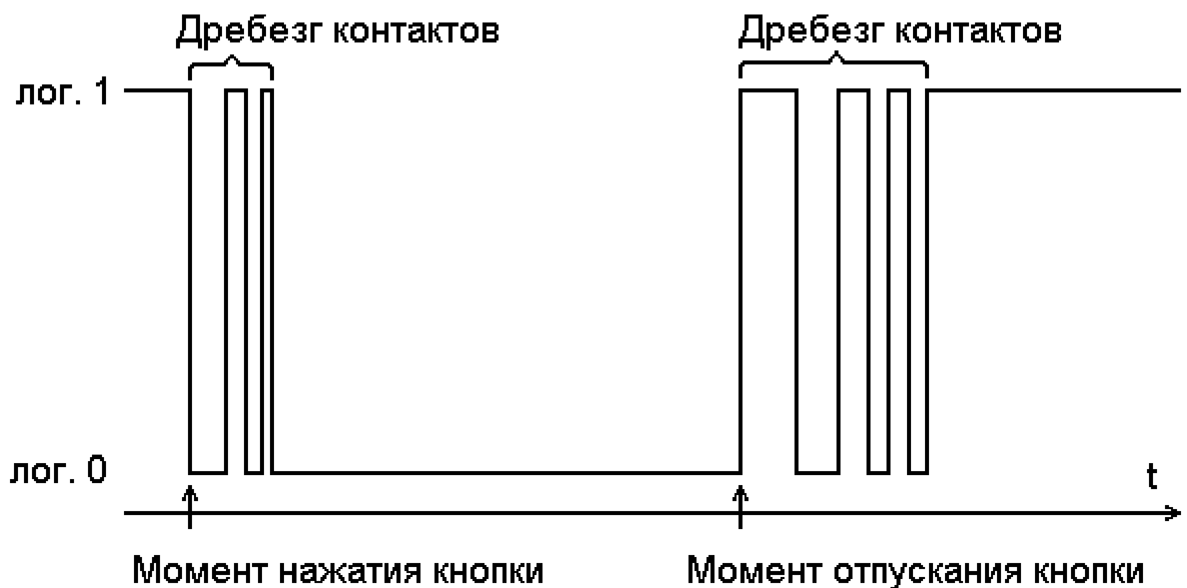

Рис 1.6

Длительность этого переходного процесса обычно составляет от сотых долей до единиц миллисекунд в зависимости от кнопки. Сам по себе дребезг представляет собой случайный процесс.

Дребезг не является проблемой, если необходимо просто отличать состояния нажатой и отжатой кнопок. Например, при решении задачи включения светодиода при зажатой кнопке.

Но он сильно усложняет задачу обнаружения момента нажатия на кнопку. Если просто отлавливать момент изменения уровня сигнала, то, как видно из рисунка, мы получим за одно нажатие хаотическое многократное нажатие и отжатие кнопки.

Алгоритм борьбы с дребезгом

Существует ряд алгоритмов, позволяющих бороться с эффектом дребезга. Самый простой из них – после первоначального обнаружения изменения уровня не отслеживать изменения уровня на выводе, к которому подключена кнопка в течение времени за которое гарантировано окончится эффект дребезга.

Этот алгоритм прост, но имеет тот недостаток, что может среагировать на шумовое изменение уровня сигнала от кнопки. Т.е. на мгновенное короткое изменение состояния сигнала. Это может произойти, например, из-за наводок с других дорожек.

Этот алгоритм обычно применяют при использовании внешних прерываний.

Другой алгоритм подразумевает накапливание информации об изменении состояния кнопки. Т.е. решение об изменении состояния принимается, если N раз подряд было получено ожидаемое значение. Например, если ожидается нажатие на кнопку, при котором на вывод микроконтроллера будет подан логический ноль, мы увеличиваем значение переменной на единицу, когда на выводе, в которому подключена кнопка, появляется ноль. Если на выводе единица, то счёт сбрасывается. Этот

алгоритм позволяет отфильтровать дребезг и принять решение только при наличии на входе МК стабильного сигнала.

USART

Общие положения

Интерфейс USART — последовательный универсальный синхронно-асинхронный приемо-передатчик. Передача данных в USART осуществляется через равные промежутки времени. Этот временной промежуток определяется заданной скоростью USART и указывается в бодах (Для символов, которые могут принимать значения, равные только нулю или единице бод эквивалентен битам в секунду). Существует общепринятый ряд стандартных скоростей: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600 бод.

Помимо бит данных USART автоматически вставляет в поток синхронизирующие метки, так называемые стартовый и стоповый биты. При приёме эти лишние биты удаляются. Обычно стартовый и стоповый биты отделяют один байт информации (8 бит), однако встречаются реализации USART, которые позволяют передавать по 5, 6, 7, 8 или 9 бит. Биты, отделённые стартовым и стоповым сигналами, являются минимальной посылкой. USART позволяет вставлять два стоповых бита при передаче для уменьшения вероятности рассинхронизации приёмника и передатчика при плотном трафике. Приёмник игнорирует второй стоповый бит, воспринимая его как короткую паузу на линии.

Принято соглашение, что пассивным (в отсутствие данных) состоянием входа и выхода USART является логическая «1». Стартовый бит всегда логический «0», поэтому приёмник USART ждёт перепада из «1» в «0» и отсчитывает от него временной промежуток в половину длительности бита (середина передачи стартового бита). Если в этот момент на входе всё ещё «0», то запускается процесс приёма минимальной посылки. Для этого приёмник отсчитывает 9 битовых длительностей подряд (для 8-бит данных) и в каждый момент фиксирует состояние входа. Первые 8 значений являются принятыми данными, последнее значение проверочное (стоп-бит). Значение

стоп-бита всегда «1», если реально принятое значение иное, USART фиксирует ошибку.

На рис 1.7 показана временная диаграмма передаваемых данных по USART.

Рис 1.7

Поскольку синхронизирующие биты занимают часть битового потока, то результирующая пропускная способность UART не равна скорости соединения. Например, для 8-битных посылок формата 8-N-1 синхронизирующие биты занимают 20 % потока, что для физической скорости 115 200 бод даёт битовую скорость данных 92160 бит/с или 11 520 байт/с.

Контроль чётности

В протоколе USART имеют возможность автоматически контролировать целостность данных методом контроля битовой чётности. Когда эта функция включена, последний бит данных («бит чётности») всегда принимает значение 1 или 0, так чтобы количество единиц в байте всегда было четным.

Управление потоком

В старые времена устройства с USART могли быть настолько медлительными, что не успевали обрабатывать поток принимаемых данных. Для решения этой проблемы модули USART снабжались отдельными выходами и входами управления потоком. При заполнении входного буфера

логика принимающего USART выставляла на соответствующем выходе запрещающий уровень, и передающий USART приостанавливал передачу. Позже управление потоком возложили на коммуникационные протоколы, и надобность в отдельных линиях управления потоком постепенно исчезла.

Область применения.

USART обычно применяется для проводной связи между двумя устройствами, расстояние между которыми не превышает единиц метров. Увеличение дальности связи приводит к зашумлению сигнала и уменьшению вероятности его правильного приёма.

Протоколы передачи данных.

Для общения между различными устройствами обычно используется стандартный формат передачи данных. Такой формат называется протоколом.

Данные обычно разбиваются на пакеты, отдельные самодостаточные наборы данных. Протокол определяет содержимое этого пакета, его интерпретацию, местоположение различных данных в пакете и т.д.

В этом методическом пособии будут рассматриваться только протоколы, описывающие непосредственное взаимодействие двух устройств. Пользователь может самостоятельно разработать протокол, который наиболее подходит под его задачи. Но существует ряд общих черт протоколов, знание которых позволяет организовывать более эффективное общение между двумя устройствами.

Пакет данных может включать в себя следующие поля: стартовая последовательность, команда, длина данных, данные, CRC. Разумеется, ими список полей не ограничивается. Поля могу занимать любое заранее оговорённое количество байт. Рассмотрим упомянутые поля подробнее.

Стартовая последовательность говорит о начале передачи. Зачастую её делают уникальной, т.е. такой, что она больше не может встречаться в пакете. Встретив её, устройство точно знает, что это начало пакета и начинает приём.

Команда говорит устройству, что ему надо делать. Это может быть команда на запрос или пересылку данных, чтение настроек и т.д. В протоколе чётко определяется список команд, которые могут быть выполнены устройством.

Зачастую в пакете передаётся не фиксированное число данных. Для того, чтобы устройство знало сколько именно данных ему передают, существует поле длина данных.

Данные содержат в себе информацию, передаваемую устройству. Это может быть всё, что угодно, кроме команды этому устройству.

CRC – это контрольная сумма. Обычно она считается последовательным сложением байт, входящих в пакет по модулю 256. При окончании приёма, устройство повторяет процедуру и, если значения совпали, значит, данные приняты правильно.

Кроме перечисленных полей, в пакете могут присутствовать другие, связанные с особенностями построения сети. Например, при количестве устройств в сети больше 2-х, зачастую вводятся поля адресов, идентифицирующих как устройство передающее данные, так и устройство, которому эти данные предназначены.

Приём данных.

Приём данных, переданных по протоколу, описанному выше, может быть осуществлён двумя способами: побайтный и приём в целом.

Побайтный приём заключается в том, что каждый принятый байт тут же обрабатывается программой. Одновременно с приёмом идёт поиск начала пакета, выявление и запись в отдельные переменные полей команды, данных и т.д. По окончании приёма производится выполнение полученной команды.

Принятые данные по прерыванию записываются в буфер приема и увеличивается счётчик количества необработанных байт. В бесконечном цикле проверяется этот счётчик и, если он отличен от нуля, производится обработка принятых байт, а счётчик необработанных данных уменьшается.

Оба процесса приёма и обработки могут идти параллельно, т.к. приём, за исключением записи в буфер, идёт аппаратно и задействует АЛУ.

Этот метод обычно используется, когда неизвестна точная длина пакета и нет никакой последовательности, которая говорит о его окончании. И, если поток данных почти непрерывный (период передачи пакетов сравним с его длительностью). Тогда надо производить обработку пакета предельно быстро. Это обеспечивается эти методом.

К достоинствам этот метода стоит отнести высокую скоростью работы и возможность обработки данных без последовательности, говорящей об окончании передачи, т.к. во время приёма программа может по уже обработанным полям выяснить фактическое количество передаваемых данных.

Недостатком данного алгоритма является его сложность и не применимость, если производится сложный программный алгоритм приёма пакета.

Приём в целом заключается в том, что данные сначала полностью принимаются и только по приёму последовательности, говорящей об окончании передачи начинается обработка пакета.

Данный алгоритм несколько проще в реализации, но требует длительного промежутка между передачами для обработки уже принятых данных.

Буферы приёма.

Принимаемые данные перед обработкой должны сохраняться в отдельный, специально созданный для этого массив. Такой массив называется буфер. Есть два способа заполнения массива: Со сбросом в начало и «круглый» буфер.

Алгоритм сброса в начало подразумевает, что после обработки принятого пакета следующий пакет опять записывается с начала массива. Этот способ прост в реализации, но менее надёжен.

Алгоритм круглого буфера подразумевает, что принятые данные расписываются с начала до конца буфера вне зависимости от принятого пакета, а затем только сбрасываются с начало. Получается запись данных по кругу. Этот алгоритм сложнее, но позволяет принимать данные непрерывно.

DMA

DMA (Direct Memory Access - прямой доступ к памяти) позволяет передавать блоки данных, минуя процессор. Т.е. раньше, если необходимо скопировать блок данных из одного буфера в другой, копирование производилось в цикле вида:

```
for (i=0; i<size; i++)
{
    A[i] = B[i];
}
```

Если size - огромное число, то копирование отнимало много времени. Если же процессор параллельно должен был опрашивать кнопки, мигать светодиодом и т.д., данные процессы останавливались на время копирования. Это неудобно и не оптимально, а зачастую может быть критично.

При использовании DMA вы просто говорите ему, что, куда и сколько копировать и говорите когда начинать. Окончив выполнения работы, DMA оповестит об этом процессор. Сам же процессор может в то время, пока DMA копирует данные выполнять другие задачи, требующие его непосредственного участия.

Другой пример - передача данных через UART. Зачастую встаёт задача передачи больших объёмов данных. К примеру, информации, снятой с датчика. Но нельзя тратить время на передачу, т.к. тогда процессор не будет снимать данные с датчика. В этом случае хорошо использовать DMA для передачи уже снятой информации. Можно даже попробовать настроить DMA, которое сначала будет снимать данные с датчика, записывая их в определённую область памяти, а по окончанию записи заданного числа данных, указать другому каналу DMA начать копировать данные из этого буфера с UART, пока первый канал параллельно заполняет другой буфер.

SPI

Обзор выводов SPI

SPI (Serial Peripheral Inteface) представляет собой интерфейс, включающий в себя протокол только физического уровня, который на физическом уровне определяет алгоритм передачи данных от одного устройства к другому. В этом смысле SPI подобен UART. Но на этом сходство заканчивается.

Для передачи данных в SPI используется 4 вывода. Назначение и название выводов приведены в табл. 2.2.

Таблица 2.1

Название	Расшифровка	Назначение
SCLK	Clock	Сигнал синхронизации передаваемых данных.
MOSI	Master Out – Slave In	Вывод для передачи данных от ведущего к ведомому.
MISO	Master In – Slave Out	Вывод для передачи данных от ведомого к ведущему.
CS	Chip Select	Вывод выбора микросхемы.

Структура SPI подразумевает, что в системе существует одно ведущее устройство и множество ведомых устройств. При этом одновременно ведущее устройство может общаться только с одним ведомым устройством. Для выбора ведомого устройства с которым будет общаться ведущее используется вывод CS (рис. 2.24).

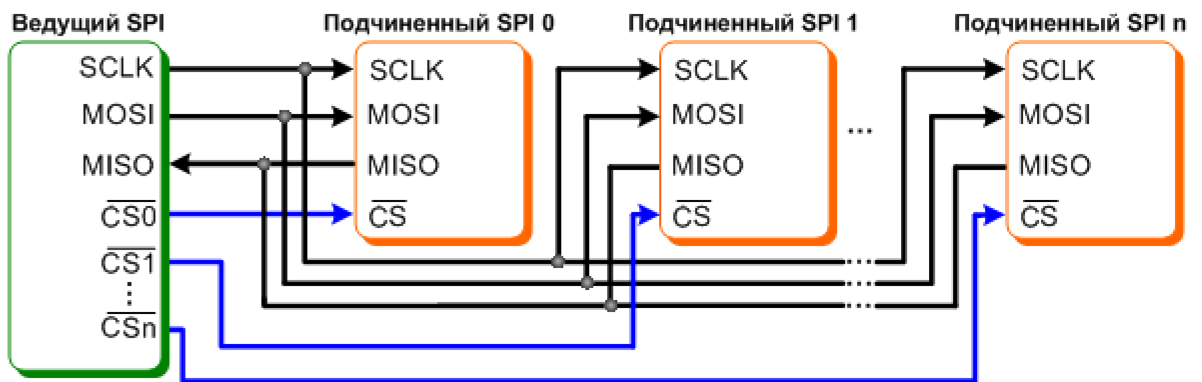

Рис. 2.24

Этот способ позволяет подключить одновременно любое число устройств на один SPI. Главное, что бы хватило выводов для выбора микросхемы.

В простейшем случае полноценное подключение выглядит как на рис. 2.25.

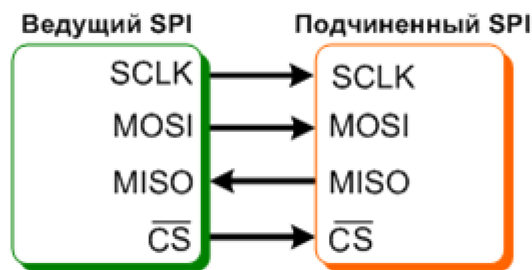

Рис. 2.25

Несмотря на то, что кажется, что выбор микросхемы в таком подключении не нужен, на самом деле многие микросхемы в своём протоколе используют выбор микросхемы для сброса буферов приёма и передачи. Теперь разберёмся с алгоритмом передачи самих данных по этому интерфейсу.

Протокол передачи

Протокол передачи по интерфейсу SPI по сути идентичен логике работы сдвигового регистра (рис 7.6), которая заключается в выполнении операции сдвига и, соответственно, побитного ввода и вывода данных по определенным фронтам сигнала синхронизации. При этом во время передачи ведущий выдаёт биты данных на вывод MOSI, а ведомый – на выод MISO.

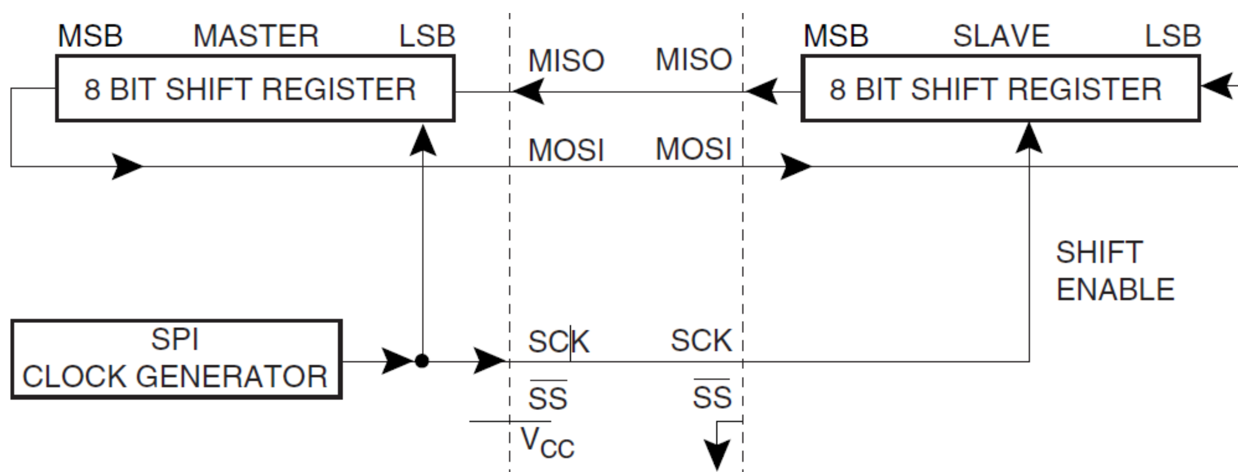

Рис 7.6

Установка данных при передаче и выборка при приёме всегда выполняются по противоположным фронтам сигнла синхронизации, который передаётся по линии CLK. Это необходимо для гарантирования выборки данных после надёжной их установки. Если при этом учесть, что в качестве первого фронта в цикле передачи может выступать нарастающий или падающий фронт, то всего возможно четыре варианта логики работы интерфейса SPI. Эти варианты получили название режимов SPI и описываются двумя параметрами:

CPOL - исходный уровень сигнала синхронизации (если CPOL=0, то линия синхронизации до начала цикла передачи и после его окончания имеет низкий уровень (т.е. первый фронт нарастающий, а последний - падающий), иначе, если CPOL=1, - высокий (т.е. первый фронт падающий, а последний - нарастающий));

CPHA - фаза синхронизации; от этого параметра зависит, в какой последовательности выполняется установка и выборка данных. Если CPHA=0, то по переднему фронту в цикле синхронизации будет выполняться выборка данных, а затем, по заднему фронту, - установка данных. Если же CPHA=1, то установка данных будет выполняться по переднему фронту в цикле синхронизации, а выборка - по заднему.

Внешний вид получающихся сигналов для каждого режимов представлен на рис 7.7 и 7.8.

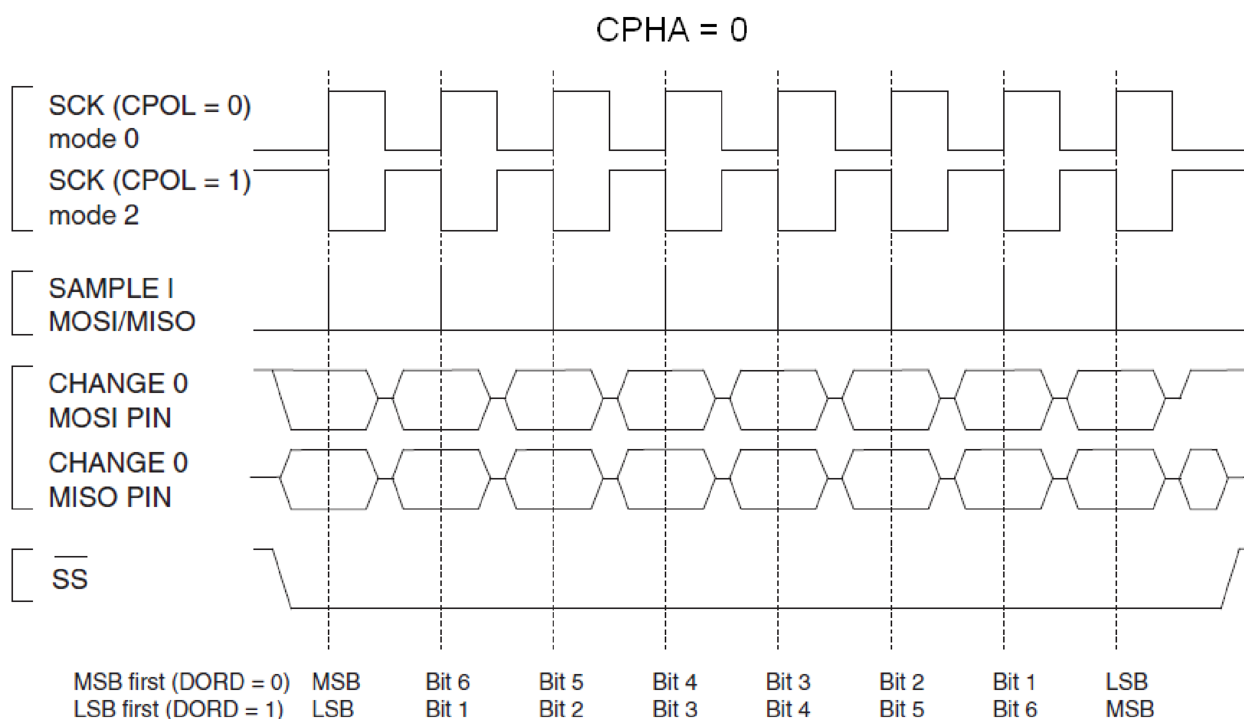

Рис 7.7

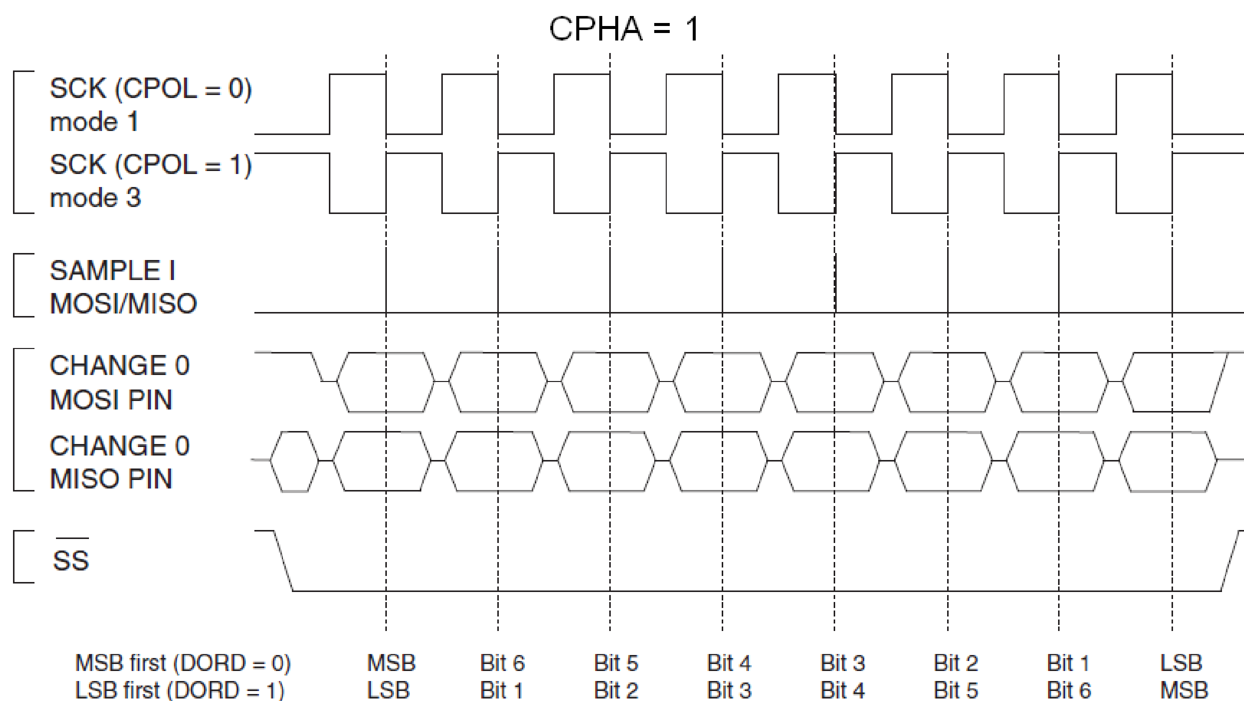

Рис 7.8

Ведущая и подчиненная микросхемы, работающие в различных режимах SPI, являются несовместимыми, поэтому, во время настройки этого интерфейса самое главное правильно установить эти два параметра. Для упрощения задачи часто в технической документации можно найти временные диаграммы, иллюстрирующие используемый режим работы

ведомой микросхемой. Остаётся только сравнить это изображение с теми, что даны выше и установить параметры, реализующие наиболее подходящую форму сигнала.

Применение SPI

Самое очевидное применение SPI – это подключение различных микросхем. Например, датчиков, с которых будут считаываться данные. При этом к одному SPI можно подключить любое число датчиков и опрашивать их по-очереди. При этом используется двунаправленная (фулдуплекс) четырёхпроводная шина, рассмотренная выше.

Но с помощью SPI можно просто устанавливать некий параметр с подключенной микросхеме. Например, при подключении ЦАП по SPI на его выходе будет появляться напряжение, пропорциональное переданному числу. В этом случае вывод MISO не используется.

Также SPI можно использовать как генератор цифровых сигналов определённого вида. Из-за того, что в SPI в отличие от UART отсутствуют служебные биты в потоке данных, можно формировать поток этих данных любого вида, реализуя физические уровни протоколов, аппаратная реализация которых в микроконтроллере не предусмотрена.

Например, с помощью SPI удобно работать со сдвиговым регистром. Тактовая SPI в этом случае является тактовой регистра, а данные, передаваемые SPI, будут по этой тактовой по-очереди записываться в этот регистр.

Но наиболее интересный пример нестандартного применения SPI – это реализация протокола, основанного на ШИМ сигналах. Т.е. когда 0 передаётся одной длительностью высокого сигнала, а 1 – другой. При этом период остаётся неизменен.

Пусть такого сигнала будет равен 8 мкс. Длительность 0 – 3 микросекунды, а 1 – 6 микросекунд. В этом случае надо настроить SPI на передачу данных на частоте 1 МГц. В этом случае передача числа 0xE0 –

передаст бит ноль, а передача числа 0xFC – передаст бит 1. При условии, что первым выходит старший бит.

Очевидно, что для передачи одного байта информации, в этом случае требуется передать 8 байт.

Жидкокристаллические индикаторы

Применение дисплеев

Жидкокристаллические индикаторы (ЖКИ) являются одними из основных средств вывода информации для цифровых систем. Они обеспечивают отображение большого объема информации при хорошей различимости контрастности и низком энергопотреблении, благодаря чему широко используются в измерительных приборах, медицинском оборудовании, промышленном оборудовании, информационных системах, аппаратуре с автономным питанием.

Дисплей WH1602 D–NGG–CT

Первое, на что стоит обратить внимание при подключении дисплея – это назначение его выводов и в частности на выводы питания. Выводы дисплея WH1602D-NGG-CT-X приведены в табл. 1.3.

Табл. 1.3

Номер вывода	Обозначение	Уровень	описание
Выводы для подключения LCD к питанию			
1	Vdd	+5В	Напряжение питания для логической схемы
2	Vss	0В	Заземление
3	V0	переменный	Напряжение питания для ЖКИ, (контрастность изображения)
Выводы для управления LCD модулем			
4	RS	H/L	Выбор регистра контроллера L(0): регистр команды H (1): регистр данных,
5	R/W	H/L	Выбор режима обмена: чтение/запись H(1): Чтение(с LCD) L(0): Запись(в LCD)
6	E	H, H/L	Разрешающий сигнал

			H(1) – запуск, L(0) – Стоп
Выводы двунаправленной шины данных			
7	DB0	H/L	Бит данных 0 (8-ми битный режим) (младший бит в 8-ми битном режиме)
8	DB1	H/L	Бит данных 1 (8-ми битный режим)
9	DB2	H/L	Бит данных 2 (8-ми битный режим)
10	DB3	H/L	Бит данных 3 (8-ми битный режим)
11	DB4	H/L	Бит данных 4 (8-ми и 4-х битные режимы) (младший бит в 4-х битном режиме)
12	DB5	H/L	Бит данных 5 (8-ми и 4-х битные режимы)
13	DB6	H/L	Бит данных 6 (8-ми и 4-х битные режимы)
14	DB7	H/L	Бит данных 7 (8-ми и 4-х битные режимы) (старший бит)

Описание контроллера дисплея

В ЖКИ встроен контроллер (БИС) HD44780, его структурная схема представлена на рис.1.8. БИС имеет два 8-битовых регистра: регистр команд (IR) и регистр данных (DR), RS – регистровый переключатель. AC - Счетчик адреса. DDRAM (Display data RAM ОЗУ ASCII-кодов отображаемых символов). Память знакогенератора (хранит матрицы начертания символов (рис.1.4) и состоит из CGROM (Character generator ROM – ПЗУ знакогенератора) и CGRAM (Character generator RAM – ОЗУ знакогенератора)).

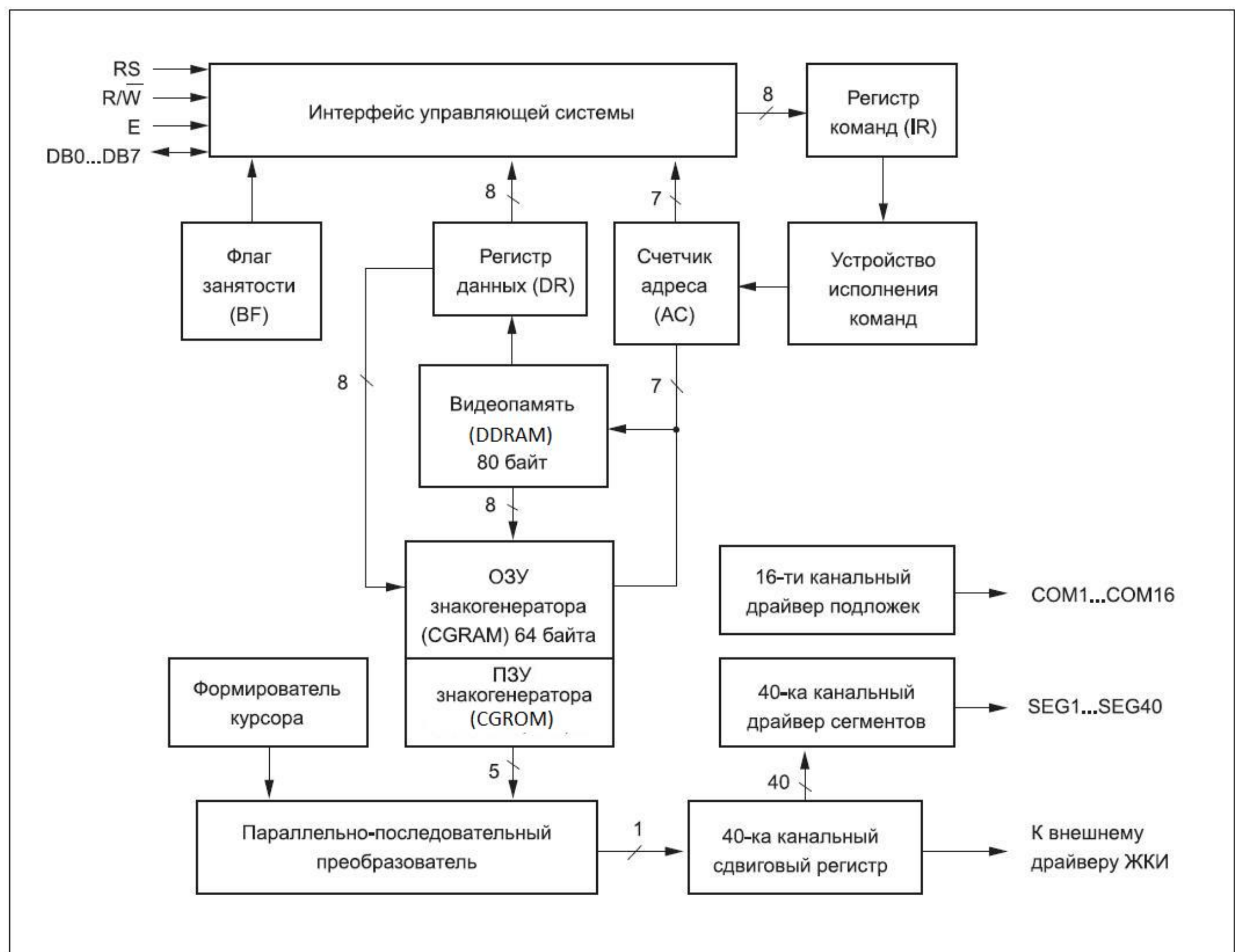

Рис. 1.8 – упрощенная структурная схема контроллера (БИС) HD44780.

IR – хранит коды операций, таких как: очистка дисплея, перемещение курсора, а также информацию об адресах памяти отображаемых данных (DDRAM) и генератора символов (CGRAM). В регистр команд можно только записывать информацию из микропроцессора.

DR – временно хранит данные, предназначенные для записи или чтения из DDRAM или CGRAM. Когда адресная информация записывается в регистр команд, данные из DDRAM или CGRAM сохраняются в регистре данных.

RS – регистровый переключатель для выбора регистров IR или DR. (RS = 1 — данные (DR), RS = 0 — команда (IR))

AC - Счетчик адреса назначает адреса и DDRAM, и CGRAM

LED A, LED K- выводы для подключения подсветки (рис.1) (в нашем случае не используются).

DDRAM. Такую память называют видеопамятью или видеобуфером. Видеобуфер в символьных индикаторах обычно содержит 80 ячеек памяти – больше, чем число знакомест дисплея. У двухстрочных индикаторов ячейки с адресами от 0x00 и до 0x27 отображаются на верхней строке дисплея, а ячейки с адресами 0x40 … 0x67 – на нижней строке. Смещая видимое окно дисплея (выделено серым фоном) относительно DDRAM, можно отображать на дисплее различные области видеопамяти. Сдвиг окна индикатора относительно видеобуфера для верхней и нижней строк происходит синхронно.

CGROM – в эту память на заводе-изготовителе загружены начертания символов таблицы ASCII. Содержимое CGROM изменить нельзя.

CGRAM – для того, чтобы пользователь смог самостоятельно задать начертание нужных ему символов (Под ячейки CGRAM отведены первые (младшие) 16 адресов таблицы кодов).

АЦП

Подключение АЦП

Прежде всего, рассмотрим подключение АЦП. Для чего нужна каждая ножка, показано в таблице 1.4.

Таблица 1.4

Название	Тип сигнала	Примечания
V_{REF+}	Вход. Положительное опорное напряжение.	Положительное напряжение $2.4\ V \le V_{REF+} \le V_{DDA}$
V_{DDA}(1)	Вход. Аналоговое питание.	Равно цифровому питанию. $2.4\ V \le V_{DDA} \le 3.6\ V$
V_{REF-}	Вход. Отрицательное опорное напряжение.	Меньшее(отрицательное) напряжение $V_{REF-} = V_{SSA}$
V_{SSA}(1)	Вход. Аналоговая земля.	Аналоговая земля равна цифровой.
ADCx_IN[15:0]	Аналоговый сигнал	21 аналоговый канал

Из перечисленных ножек интересны -Vref и +Vref. Они определяют диапазон напряжений, воспринимаемых АЦП. Если подключить -Vref к земле, а +Vref к питанию, то АЦП сможет оцифровать аналоговые сигналы во всём диапазоне от 0, до питания. Т.к. питания МК составляет 3,3В, а разрядность АЦП равна 12-ти, т.е. мы имеем 2^12=4096 уровней квантовая, шум АЦП составит 3,0/4096=0,73 мВ.

В микроконтроллере stm32f4xx находится 3 независимых АЦП, каждое из которых может оцифровать сигнал с 21 канала. Какое АЦП к какому выводу микроконтроллера подключено можно узнать из технической документации pinout.

Виды АЦП.

В микроконтроллерах фирмы ST существует 2 вида каналов АЦП: регулярные и инжекторные. Эти 2 канала настраиваются независимо. Но работать может только один из них для каждого канала. Основным различием этих каналов является то, что для хранения данных, получаемых с помощью регулярного канала используется только один регистр. Это не плохо, если вам надо снять за один раз данные только с одного канала для

каждого АЦП. Но, если Вам надо производить сканирование данных, то все снятые данные будут записываться с один и тот же регистр. Т.о. при чтении данных в прерывании по окончании преобразования Вы будете получать только последние снятые данные. Эту проблему призваны исправить инжекторные каналы. У них предусмотрены 4 регистра для хранения данных. Т.е. Вы сможете хранить данные с 4-х каналов сканирования. Недостатком инжекторных каналов является несколько более сложная система настройки, в которой надо описать данные, с какого канала в какой регистр будут записаны.

Датчик температуры LM35

Датчик температуры преобразует температуру чувствительного элемента в напряжение по линейному закону.

Температурный датчик LM35 имеет 3 вывода: Питание, земля и данные (см. рис 1.10).

Рис. 1.10

В качестве питания можно использовать напряжение от 4 до 20 В. Выходом датчика является аналоговый сигнал, пропорциональный температуре этого датчика. Причём на 1 градус Цельсия приходится 10мВ. Например, если на выходе 2 В, из этого следует, что температура датчика 200 град. Цельсия.

Пересчёт значений АЦП в напряжения

Для пересчёта значения, полученного с АЦП, прежде всего надо учитывать, что за ноль принимается нижняя граница преобразования, а за максимальное значение – верхняя. Т.к. для МК stm32f4 в данном случае нижней границей является земля (ноль), а верхней – напряжение, равное 3 В. Получаем, что максимальное значение напряжения соответствует числу 0x0FFF = 4095. Число 0x0FFF – это 12-тибитное преобразование. 4095 – это 2 в степени 12 без 1.

На рис. 1.11 показаны обе шкалы. Слева шкала напряжений, справа – значений с АЦП.

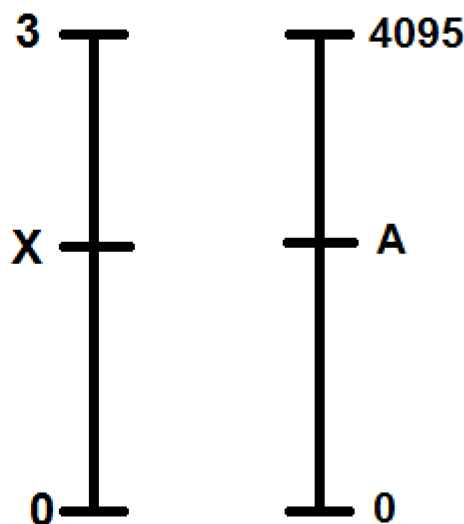

Рис. 1.11

Пусть было получено значение A с АЦП. Этому значению соответствует значение X, которое является реальным напряжением, которое было в этот момент подано на вывод АЦП. Очевидно, что 4095/3 = A/X. Отсюда находится X в напряжениях. Но лучше считать в сразу милливольтах. Тогда то же соотношение будет выглядеть так: 4095/3000 = A/X

Остаётся только перевести напряжения в температуру, как описано выше.

Цифро-Аналоговый преобразователь

Цифро-аналоговое преобразование позволяет преобразовать цифровой сигнал в соответствующий ему аналоговый сигнал. Микроконтроллер stm32f4xx имеет в своём распоряжении 2 одноканальных 12-тибитных ЦАП.

Акселерометр

Принцип работы акселерометра.

Акселерометр предназначен для снятия значения ускорений и превращения их в электрические сигналы. Принцип работы акселерометра представлен на рис 1.12.

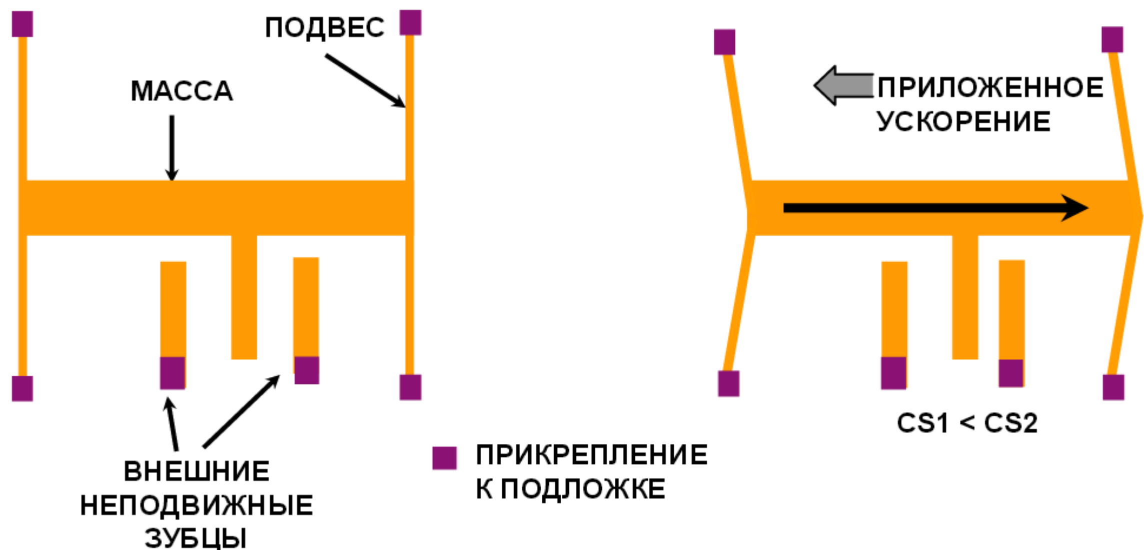

Рис 1.12

Подвижная масса подвешена на неподвижной подложке. Между зубцами возникает ёмкость. Именно она и измеряется для получения значения приложенного ускорения. Когда датчик находится в неподвижном состоянии, ёмкость сохраняет постоянное значение. Обычно при этом центральная пластина находится посередине между двумя крайними. Если на датчик воздействует сила, появляется ускорение и пластина смещается. По изменению ёмкости можно определить величину ускорения.

Обычно выпускаются акселерометры с одним из двух типов выходов: аналоговым или цифровым. Аналоговый выход полностью повторяет ускорения, с которыми двигается датчик. Для снятия данных с этого датчика достаточно оцифровать эти данные АЦП.

Цифровой выход так же обычно бывает двух типов: SPI и I2C. Причём зачастую у одного и того же датчика возможны оба варианта подключения. При этом датчик имеет некоторую цифровую часть, включающую в себя

АЦП, регистры данных, регистры управления и ряд других регистров, определяющих дополнительные функции датчика. Например, датчик может переходить из спящего режима в режим снятия данных по порогу автоматически. Чтобы снять данные с такого датчика, надо отправить запрос на чтение регистра, куда записываются нужные значения. В режиме SPI при следующем запросе, вы получите запрошенные значения.

Акселерометр LIS302DL

Акселерометр LIS302DL имеет как выход SPI, так и I2C и может быть подключен по любому из них. Внутри его встроено 8-ми разрядное АЦП. Датчик поддерживает 2 диапазона работы: 2 и 8g. Максимальная поддерживаемая частота обновления данных: 400 Гц.

Для включения обновления датчика в регистр по адресу 0x20 надо бит PD установить 1.

Акселерометр LIS3DSH

Акселерометр LIS3DSH имеет цифровой выход и поддерживает интерфейсы SPI и I2C. Подключиться можно с помощью любого из них. Акселерометр поддерживает 5 диапазонов работы: 2,4,6,8,16g и имеет 16-ти разрядные АЦП и, как следствие, 16-ти разрядные выходные значения по каждой из трёх осей.

Для начала работы с акселерометром надо в регистр по адресу 0x20 записать в старшие 4 разряда, значение отличное от 0. Т.к. если записан 0, датчик находится в режиме сна. Остальные значения устанавливают частоту обновления данных в диапазоне от 3 до 1600 Гц (рис 1.13).

ODR3	ODR2	ODR1	ODR0	ODR selection
0	0	0	0	Power down
0	0	0	1	3.125 Hz
0	0	1	0	6.25 Hz
0	0	1	1	12.5 Hz
0	1	0	0	25 Hz
0	1	0	1	50 Hz
0	1	1	0	100 Hz
0	1	1	1	400 Hz
1	0	0	0	800 Hz
1	0	0	1	1600 Hz

Рис 1.13

Инкрементальный энкодер

Принцип работы

Инкрементальные энкодеры (рис 1.14) генерируют последовательный импульсный цифровой код, содержащий информацию относительно угла поворота. Если энкодер не вращается, то останавливается и передача импульсов. Основным рабочим параметром датчика является количество импульсов за один оборот. Мгновенную величину угла поворота объекта определяют посредством подсчёта импульсов от старта. Выходной сигнал имеет два канала, в которых идентичные последовательности импульсов сдвинуты на 90° относительно друг друга (парафазные импульсы), что позволяет определять направление вращения.

Рис 1.14

Применяется энкодер обычно либо для контроля поворота вала, либо для изменения каких-либо значений. В дальнейшем мы рассмотрим именно второй вариант применения энкодера.

Выводы энкодера pec16-4220f-s0024

Энкодер pec16-4220f-s0024 имеет 5 выводов. 2 вывода, расположенные с одного края, отвечают за кнопку. При нажатии на энкодер эти выводы замыкаются друг на друга. Расположенные с другой стороны три вывода отвечают за поворот. Из которых на выводах A и B (рис 1.15) появляются импульсы при повороте энкодера, в вывод C подключается к земле.

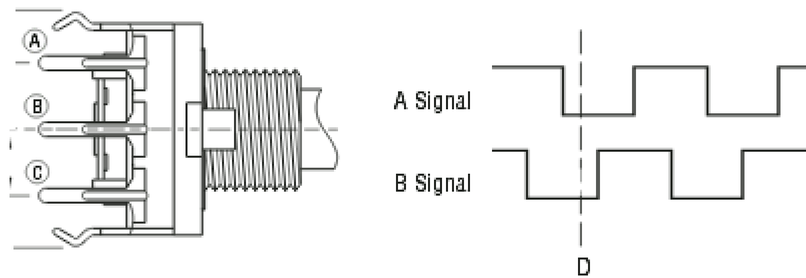

Рис 1.15

При подключении выводов A и B требуется подтянуть и резистором к питанию, т.к. при повороте энкодера выводы в определённый момент подключаются к земле, создавая тем самым импульсы. В статичном положении на выводы энкодера не подключены ни к чему. Т.е., если выводы подтянуты к питанию, на них будет логическая единица.

Клавиатура

Для подключения клавиатуры формата 4х4 достаточно одного порта микроконтроллера.

В приведённом примере столбцы клавиатуры подключены к младшей тетраде порта А микроконтроллера - выводы PA0...PA3, которые должны быть настроены на выход (см. рис 1.16).

Старшая тетрада порта - строки матрицы, настраиваются как входы и притягиваются к питанию внутренними pull-up резисторами или внешними номиналом 10 кОм.

Для того чтобы избежать конфликта логических уровней при случайном нажатии двух кнопок, принадлежащих разным столбцам матрицы, цепи 1, 2, 3 и 4 необходимо подключить к порту через резисторы номиналом 270 Ом.

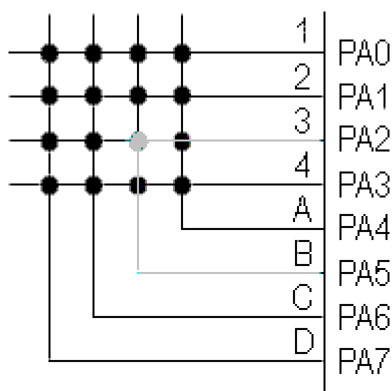

Рис 1.16

Обнаружение нажатой кнопки проводится по следующему простому алгоритму:

- исходное положение - все столбцы матрицы (PA0...PA3) в состоянии лог. 1;

- выбор столбца 1 установкой низкого уровня PA0=0;

- считывание старшей тетрады порта (строк клавиатуры) для обнаружения нажатых кнопок столбца 1 (1A, 1B, 1C или 1D). В случае

замыкания одной из указанных клавиш нулевой уровень столбца появиться на соответствующей строке матрицы.

- возврат в исходное положение - все столбцы в высоком состоянии;

-далее происходит аналогичная выборка оставшихся столбцов с последующим чтением строк матрицы.

Пример: при нажатии кнопки с адресом 3В, после выборки столбца 3 (PA2=0) на строке В будет считан низкий уровень (PA5=0).

Память

Общие положения.

Зачастую возникает необходимость сохранять ряд данных в энергонезависимую память. Такими данныхми например могут быть настройки программы для МК. Если нет внешней памяти, то приходится использовать внутренние ресурсы МК. К сожалению stm32f4 не имеет в своём составе памяти EEPROM, позволяющей писать побайтно, как в ОЗУ. Потому приходится использовать Flash. Тот самый, в который пишется и сама программа. Потому работа с этой памятью требует особой осторожности. Если её не соблюдать, то вы можете легко затереть часть (а то и всю) вашей программы и устройство перестанет работать.

Чтобы этого не произошло, в микроконтроллере stm32f4 предусмотрена защита. Для доступа к записи в эту память необходимо ввести 8-мибайтный код. Если код введён не верно, доступ к flash можно будет открыть только после сброса питания с МК. Рассмотрим основные функции flash.

Структура Flash

Flash память МК делится на сектора. Каждый сектор отвечает за заданную область памяти. Разделение памяти на сектора можно найти в технической документации. Например, в микроконтроллере stm32f407 память разделена на 12 секторов. Какой сектор, за какую область памяти в данном микроконтроллере отвечает можно узнать из приведённого ниже списка:

0. 0x08000000-0x08003FFF (16 кБ)
1. 0x08004000-0x08007FFF (16 кБ)
2. 0x08008000-0x0800BFFF (16 кБ)
3. 0x0800C000-0x0800FFFF (16 кБ)
4. 0x08010000-0x0801FFFF (64 кБ)
5. 0x08020000-0x0803FFFF (128 кБ)

6. 0x08040000-0x0805FFFF (128 кБ)
7. 0x08060000-0x0807FFFF (128 кБ)
8. 0x08080000-0x0809FFFF (128 кБ)
9. 0x080A0000-0x080BFFFF (128 кБ)
10. 0x080C0000-0x080DFFFF (128 кБ)
11. 0x080E0000-0x080FFFFF (128 кБ)

Стирать память можно только по секторам, т.е., если вам требуется записать 1 байт, вам надо куда-то скопировать весь сектор, стереть его и записать в него старые данные, заменив в них нужный байт.

Т.к. данная память является flash памятью типа NAND записать данные поверх текущих невозможно потому, что запись производится побитовым AND. Т.е. фактически это выглядит так:

ПАМЯТЬ=ПАМЯТЬ AND data, где data - ваши данные.

Данный способ записи аппаратно реализован в самой памяти. Программисту просто требуется указать, что и куда писать. По причине такого способа записи всегда перед записью одних данных поверх других необходимо стереть предыдущие, т.к. значение в ячейке можно изменить только из 1 в 0, но не обратно. Стирание же устанавливает значение всех ячеек в 1.

Рассмотрим пример: Пусть имеется ячейка памяти, в которой записано значение 0xFF, т.е. все единицы. Пусть требуется записать значение 0x56. Получаем:

A=0xFF&0x56=0x56

Теперь в данную ячейку запишем число 0x13.

A=0x56&0x13=0b01010110&0b00010011=0b00010010=0x12

Т.е. вместо 0x13 записалось значение 0x12. Если бы в ячейке были бы только нули, то ничего не записалось бы совсем.

Генератор случайных чисел.

Общие понятия

В большинстве МК присутствует функция генератора случайных чисел, но обычно он реализован следующим образом. Создана длинная псевдослучайная последовательность чисел с равномерным распределением. При запросе случайных чисел, те просто берутся по очереди из этой последовательности. Очевидно, что, если просто брать данные с начала последовательности, они всегда будут одинаковые. Чтобы этого не происходило, перед запросом случайного числа указатель на запрашиваемое число переносится вглубь последовательности на основе какого-либо быстро изменяющегося числа. Например, текущего времени. Недостатки этого метода очевидны. Полученные числа не являются случайными в прямом смысле этого слова, а только кажутся таковыми.

По счастью, в микроконтроллере существует множество вариантов создать действительно случайные числа. Самый просто способ, который проходит на ум, это считывание шумового сигнала с помощью АЦП. Именно так и сделан генератор случайных чисел в микроконтроллере stm32f4xx. Он оцифровывает усиленный тепловой шум резистора. Такой алгоритм гарантирует случайность полученного числа.

Плотность вероятности случайного процесса

Функция плотности вероятности показывает вероятность попадания значения случайной величины в некий промежуток Δx. Интеграл в бесконечных пределах от этой функции равен 1. Т.е. вероятность того, что случайная величина примет какое-то значение от $-\infty$ до $+\infty$ равна 1.

Плотность вероятности может иметь произвольный вид с учётом некоторых ограничений. Но чаще всего имеют дело с некоторыми

классическими видами плотности вероятности, среди которых стоит отметить равномерное и нормальное распределения.

При равномерном распределении случайная величина может принять любое значение с равной вероятностью. В итоге плотность распределения вероятности будет выглядеть как показано на рис 1.12.

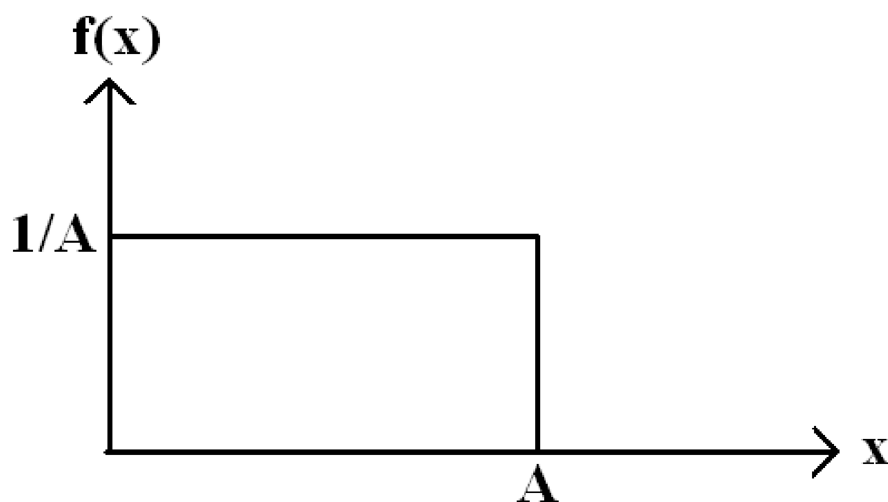

Рис 1.12

Нормальное распределение, также называемое распределением Гаусса — распределение вероятностей, которое в одномерном случае задается функцией плотности вероятности, совпадающей с функцией Гаусса:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Вид нормального распределения представлен на рис 1.17

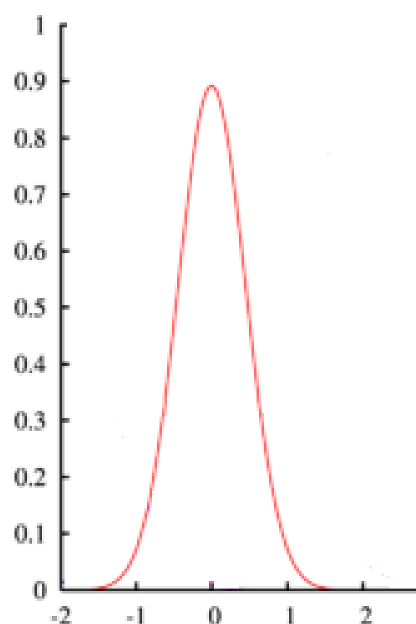

Рис 1.17

Вид нормального распределения зависит от двух параметров: математического ожидания μ и дисперсии σ^2. Первая характеризует в общем случае среднее значение случайной величины. Т.к. нормальное распределение является симметричным относительно центральной точки, то для него мат. ожидание характеризует положение максимума.

Дисперсия же характеризует разброс значений. Если дисперсия мала, то и принимаемые случайным процессом значения будут лежать вблизи мат. ожидания.

Центральная предельная теорема

Центральная предельная теорема утверждает, что бесконечная сумма слабо зависимых или независимых случайных величин с одинаковым распределением и конечной дисперсией имеет распределение близкое к нормальному.

Часть 2: Программирование микроконтроллера stm32f407vg

Основные библиотеки для микроконтроллеров stm32fxxx

Библиотека CMSIS

В микроконтроллере все управляющие регистры расположены по определённым адресам. Чтобы не держать в голове, по какому адресу, какой регистр расположен, была написана библиотека CMSIS. Эта библиотека сопоставляет адреса регистров с определёнными названиями, говорящими о назначении того или иного регистра.

Библиотека StdPeriph

Библиотека StdPeriph создана для упрощения работы с периферийными устройствами МК. Эта библиотека позволяет производить настройку нужного периферийного устройства, не обращаясь к управляющим регистрам напрямую. Всё, что требуется от программиста, это заполнить соответствующую структуру параметров и вызвать функцию инициализации с входным параметром в виде этой структуры. Это позволяет производить настройку системы, не обращаясь к технической документации на используемый микроконтроллер.

Ещё одним плюсом использования данной библиотеки является то, что она уменьшает зависимость написанного программистом кода от используемого микроконтроллера, т.к. выходные функции библиотеки по возможности одинаковы для любого микроконтроллера.

Стоит отметить, что функции библиотеки StdPeriph основаны на библиотеке CMSIS и без неё библиотека не работает. Отсюда следует один недостаток этой библиотеки. Программа, написанная с её помощью, работает зачастую существенно медленнее, чем написанная только с использованием CMSIS.

Обычно имеет смысл сделать начальную инициализацию на StdPeriph, а в случае необходимости срочной перенастройки, использовать CMSIS. Т.е. необходимо свободное владение обоими библиотеками.

Создание проекта в keil uVission

После установки и запуска программы, мы получаем окно следующего вида, как показано на рис 2.1.

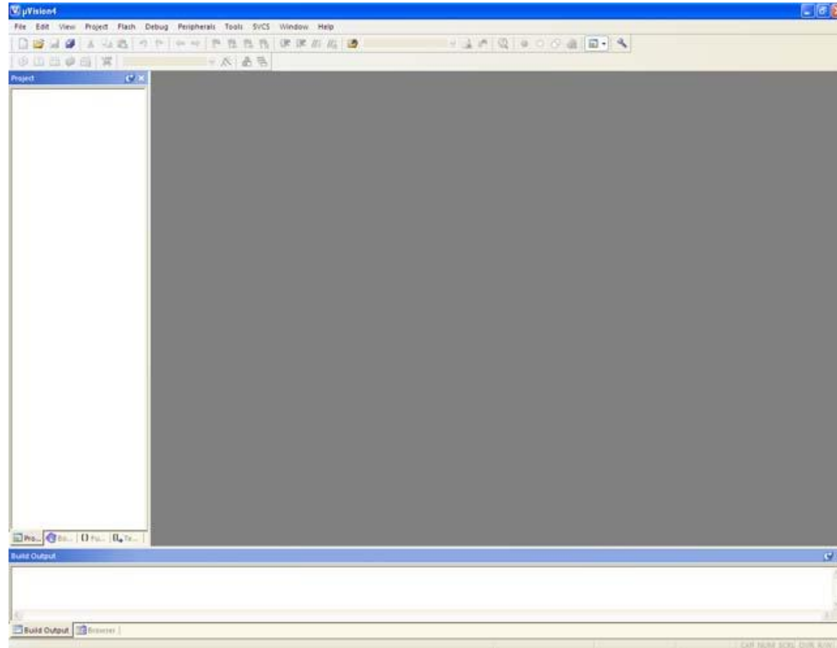

Рис 2.1

Для начала создания проекта жмём *Project -> New uVision project*. Появится окно для сохранения созданного проекта. Придумайте ему осмысленное название и поместите в отдельную папку. Жмите «Сохранить». Появится окно выбора микроконтроллера как на рис 2.2:

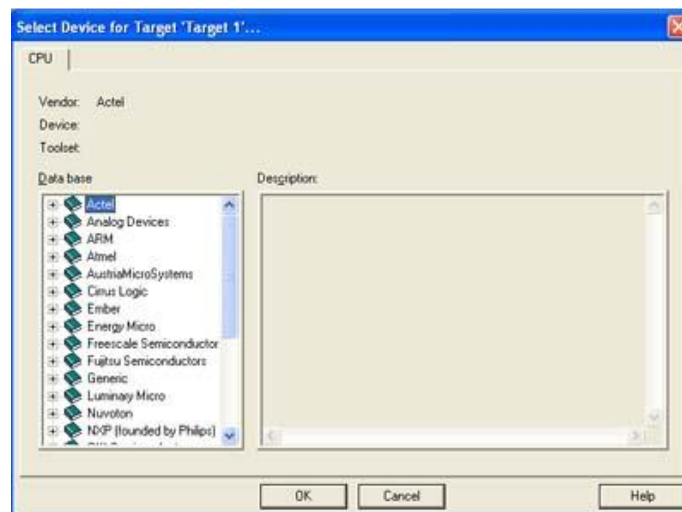

Рис 2.2

Выбираем требуемый микроконтроллер (название микроконтроллера можно прочитать на нём). Нажимаем «Ок». Появится окно с вопросом, создать ли стартовый файл (Рис 2.3). В стартовом файле прописаны начальные настройки микроконтроллера и производится выхов функции main. Нажимаем «Да».

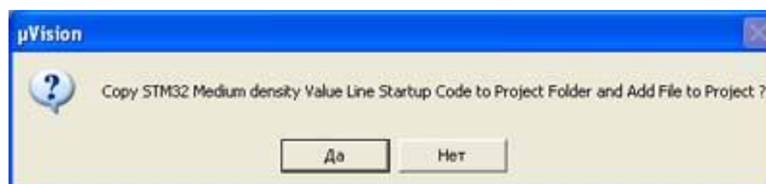

Рис 2.3

В итоге проект должен принять вид, похожий на то, что представлено на рис 2.4.

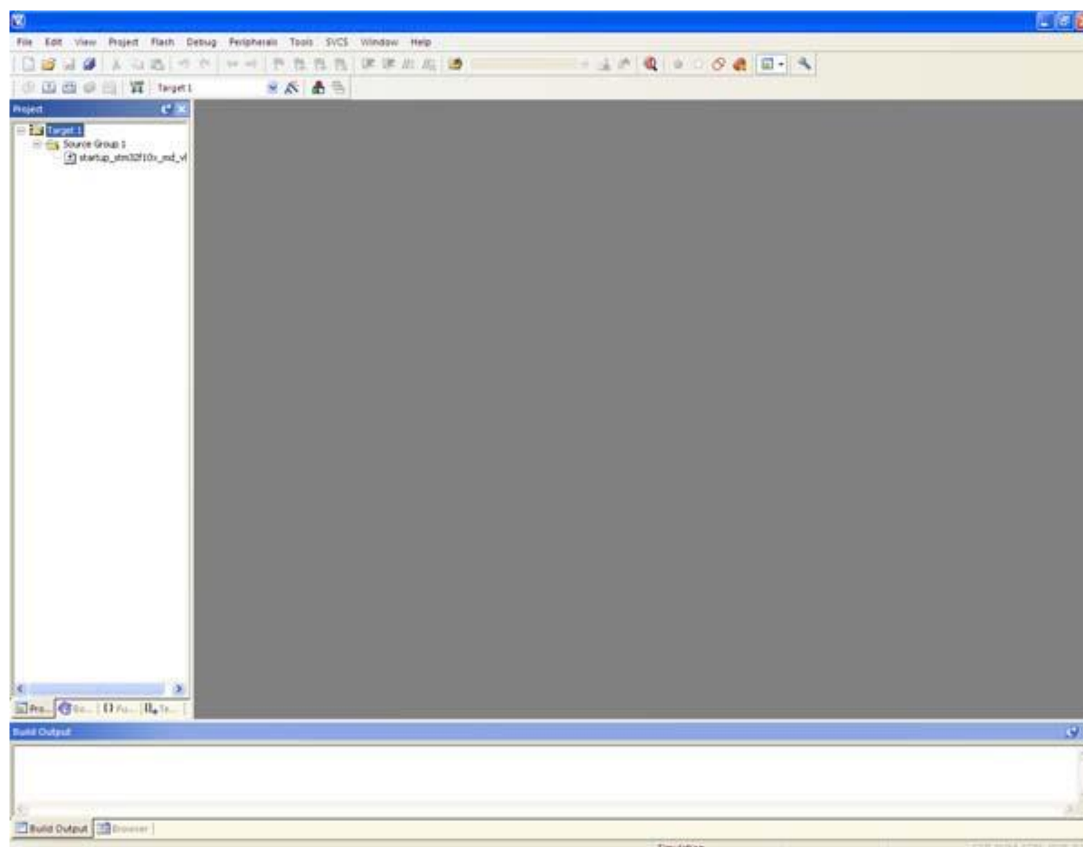

Рис 2.4

Обратите внимание на вкладку «Target 1» слева. Там хранятся файлы проекта. Здесь же вы будете подключать все необходимые библиотеки. Прежде всего подключим пользовательский файл main.c. Для этого в keil слева (окно project) щелкаем по «Target 1», выбираем «Add Group». Появится новая группа. Переименовываем её в User. В этой группе вы будете хранить свои файлы.

Далее нажимаем File -> New. Создастся новый документ. Сохраните его в заранее созданной папке User в папке вашего проекта, назвав файл main.c. Добавьте только что созданный файл в группу User.

Теперь щёлкните правой кнопкой мыши по Target1 -> Options for Target 'Target 1'. Во вкладке C/C++ пропишите в строке Include paths путь к вашему файлу main.c. По этому пути программа будет искать все упомянутые файлы. Если у вас будут лежать файлы в других местах, то пути к ним так же придётся прописать тут.

В файле main.c пропишите следующие строки:

```
#include "stm32f4xx.h"

int main(void)
{
	while(1){}
}
```

В первой строке вы подключаете файл из библиотеки CMSIS, которая ещё не подключена.

Подключение библиотеки CMSIS

Библиотека CMSIS входит в состав библиотеки StdPeriph о которой будет рассказано ниже. Скачать обе библиотеки можно с официального сайта фирмы производителя микроконтроллеров www.st.com.

Для подключения этой библиотеки скопируйте в папку с проектом всю папку с библиотекой CMSIS из архива. Папка CMSIS лежит в папке Libraries.

Для уменьшения занимаемого места на диске все папки, кроме Include и Device, можно удалить.

В программе создайте группу CMSIS и добавьте туда файл system_stm32f4xx.c, расположенный по адресу:

CMSIS\Device\ST\STM32F4xx\Source\Templates.

Во вкладке C/C++ в строке Include paths пропишите пути:

\CMSIS\Device\ST\STM32F4xx\Source

\CMSIS\Device\ST\STM32F4xx\Include

Примечание: В относительном пути к файлу не должно быть пробелов. Программа игнорирует всё, что написано после пробела и не сможет найти путь к нужным файлам.

Но перед компиляцией проект требуется настроить.

Настройка библиотеки CMSIS

Прежде всего, надо библиотеке указать тип используемого микроконтроллера.

Но прежде чем изменять библиотечные файлы, надо с них снять галочки только для чтения. Для этого найдите в папке CMSIS по адресу CMSIS\Device\ST\STM32F4xx\Include файлы stm32f4xx.h и system_stm32f4xx.h. И, зайдя в свойства, снимите эти галочки. Теперь файлы можно изменять из-под keil.

Откройте файл stm32f4xx.h. Это можно сделать из-под keil, нажав правую кнопку мыши в строке #include “stm32f4xx.h” на названии файла и в выпавшем контекстном меню выбрав open stm32f4xx.h.

Чтобы указать тип микроконтроллера, найдите в открытом файле строку:

/* #define STM32F40_41xxx */

Требуется раскомментировать эту строку, чтобы указать, что именно этот тип микроконтроллера используется в лабораторной работе. Получится следующее:

```
#define STM32F40_41xxx
```

Найдите чуть ниже строку:

```
#define HSE_VALUE ((uint32_t)xxxxxxxx),
```

где xxxxxxxx это значение тактовой частоты используемого кварцевого резонатора в Гц. Установите её равной частоте используемого тактового резонатора (см. отладочную плату). Значение тактовой частоты указано на кварцевом резонаторе на крышке.

Теперь щёлкните правой кнопкой мыши по Target1 -> Options for Target 'Target 1'. В разделе Target установите значение Xtal равным HSE.

Нажмите F7. Проект должен завершить компиляцию успешно.

Подключение библиотеки StdPeriph

Для подключения этой библиотеки, скопируйте в папку с проектом папку STM32F4xx_StdPeriph_Driver, находящуюся в той же папке Libraries. В keil создайте группу StdPeriph и добавьте в неё все файлы из папки src.

В итоге у вас должно получиться что-то похожее на то, что изображено на рис 2.5:

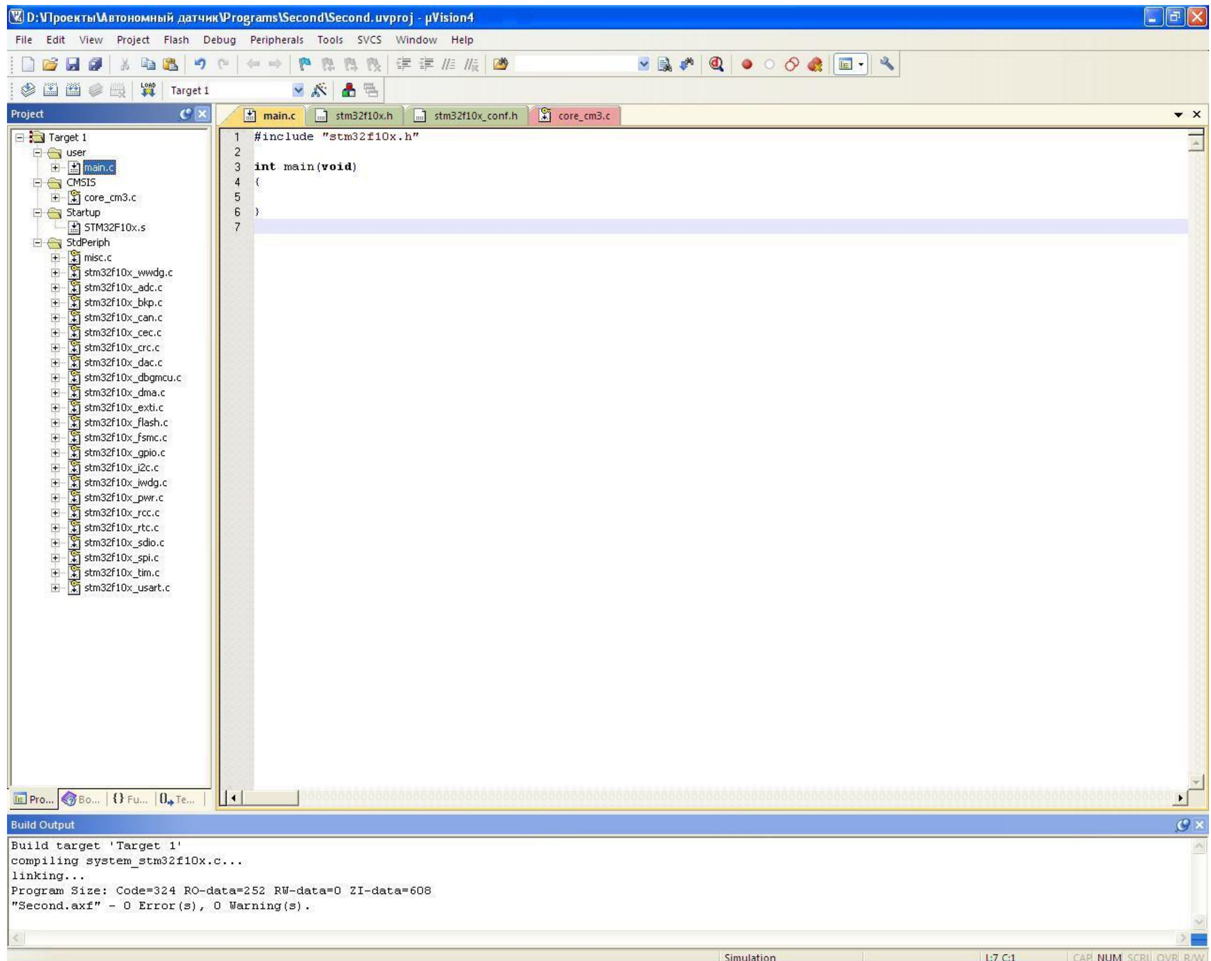

Рис 2.5

Во вкладке C/C++ в строке Include paths прописываем пути к папкам inc и src и к самой папке stdPeriph. Например:

\STM32F4xx_StdPeriph_Driver\inc

\STM32F4xx_StdPeriph_Driver\src

У вас должно получиться как на рис 2.6:

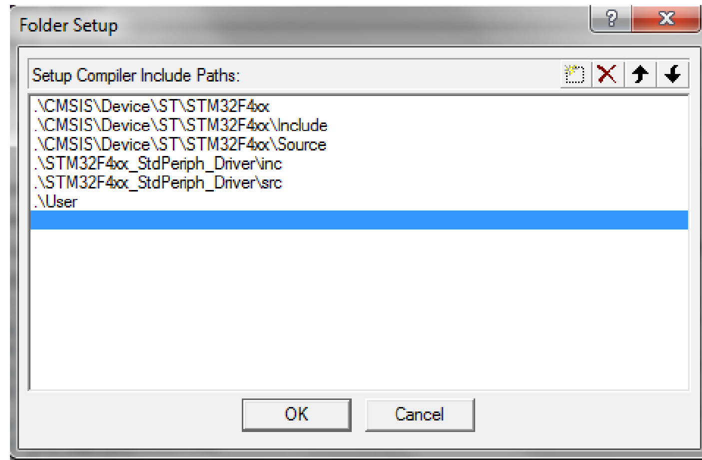

Рис 2.6

В файле stm32f4xx.h найдите строки:

```
#if !defined  USE_STDPERIPH_DRIVER
/**
 * @brief Comment the line below if you will not use the peripherals drivers. In this case, these drivers will not be included and the application code will be based on direct access to peripherals registers
 */
 /*#define USE_STDPERIPH_DRIVER*/
#endif
```

Раскомментируйте строку /*#define USE_STDPERIPH_DRIVER*/. Это значит, что вы собираетесь использовать библиотеку STDPERIPH.

Теперь добавьте в проект файл stm32f4xx_conf.h. Этот файл можно найти в папке Project\examples в любом проекте. Например, GPIO. Бросьте его в папку STM32F4xx_StdPeriph_Driver. В ней его будет просто найти в случае необходимости. Пропишите путь к папке, где лежит этот файл. В этом файле подключаются все файлы библиотеки. Неиспользуемые модули можно комментировать.

Примечание: В случае, если функции какой-то периферии не работают, имеет смысл проверить не закомментирована ли строчка с её подключением.

На этом подключение библиотеки закончено. Если нажать на F7, то проект должен собраться без ошибок.

В случае возникновения ошибок удалите файл stm32f4xx_fmc.c. Микроконтроллер stm32f407 не поддерживает эту периферию.

Настройка тактирования микроконтроллера

Микроконтроллер тактируется либо от внешнего, либо от внутреннего генератора с частотой максимум 24 МГц. Но ядро тактируется от 168 МГц. Поэтому входной тактовый сигнал проходит через модуль PLL, который умножает и делит тактовую частоту на заданные значения. Кроме того, PLL может создавать сразу несколько линий тактирования для независимого тактирования разных модулей микроконтроллера.

Для настройки тактирования ядра микроконтроллера необходимо произвести настройку этого PLL. Схема тактирования микроконтроллера stm32f407vg представлена на рис. 2.7. Функции по работе с тактированием расположены в разделе RCC.

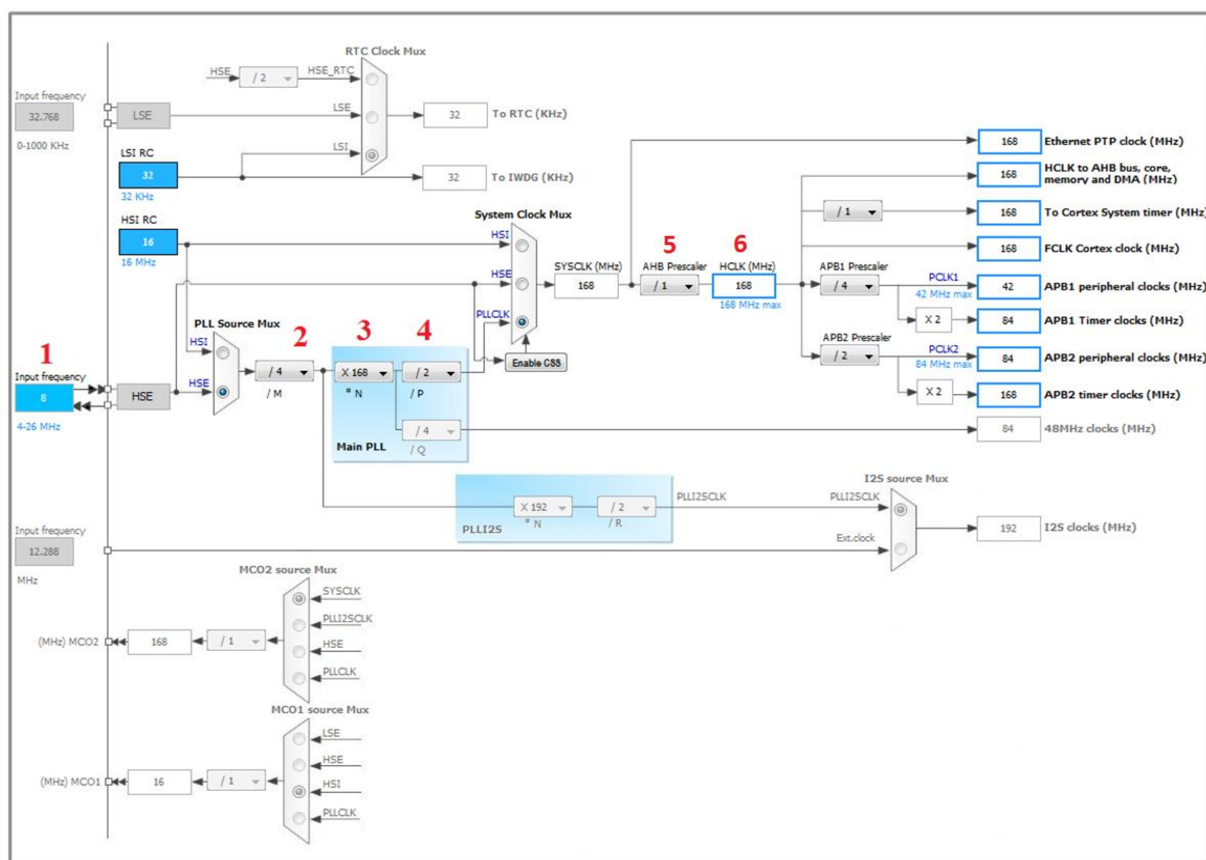

Рис. 2.7

Под номером 1 расположен кварцевый резонатор, подключаемый к микроконтроллеру. Преобразование частоты тактирования делится на три этапа. Первый – деление частоты делителем M (номер 2). На выходе должна получиться частота не менее 1 МГц. Второй – умножение умножителем N

(номер 3). И третье – деление делителем P (номер 4). Есть ещё дополнительный делитель под номером 5. В результате на выходе PLL оказывается сигнал с заданной частотой (номер 6). Следующие делители определяют частоту тактовых шин, от которых тактируются другие модули микроконтроллера, такие как таймеры.

Описанные выше параметры настраиваются в файле system _stm32f4xx.c в строка:

```
/* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
#define PLL_M      8
#define PLL_N      336
/* SYSCLK = PLL_VCO / PLL_P */
#define PLL_P      2
```

Если требуется установить тактовую частоту для шин APB1 и APB2 необходимо найти функцию SetSysClock.

В строке находятся делители как показано ниже:

```
/* PCLK2 = HCLK / 2*/
RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;
/* PCLK1 = HCLK / 4*/
RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;
```

Эти частоты важны при программировании таймеров.

Остальные блоки относятся либо ко внутреннему генератору, либо к линиям тактирования модулей, таких как USB.

Определение тактовой частоты ядра

Чтобы узнать тактовую частоту ядра, используется функция SystemCoreClockUpdate();

Она обновляет значение переменной SystemCoreClock, в которую записывается значение тактовой частоты в Гц.

Посмотреть значение этой переменной можно влибо, наведя на неё курсор, либо в watch window.

Программирование платы stm32f4-discovery

После написания кода программу необходимо собрать нажав либо на F7, либо на кнопку 2 или 3, показанные на рис 2.8.

Рис 2.8

Кнопка 1 (рис 2.8) проверяет синтаксис одной страницы, кнопка 2 добавляет в собранный проект изменения. Кнопка 3 – собирает проект целиком с начала. Т.о. при частом изменении проекта стоит использовать кнопку 2, т.к. она собирает проект значительно быстрее.

Программирование платы производится кнопкой 4. Но до использования этой кнопки требуется задать тип используемого программатора. По-умолчанию стоит симулятор.

Для изменения программатора зайдите в настройки target1. Выберите вкладку debug. Переставьте точку с Use simulator на Use … В выпадающем окне выберете St-Link и нажмите настройки. Там надо выбрать SWD вместо JTAG (см. рис 2.9).

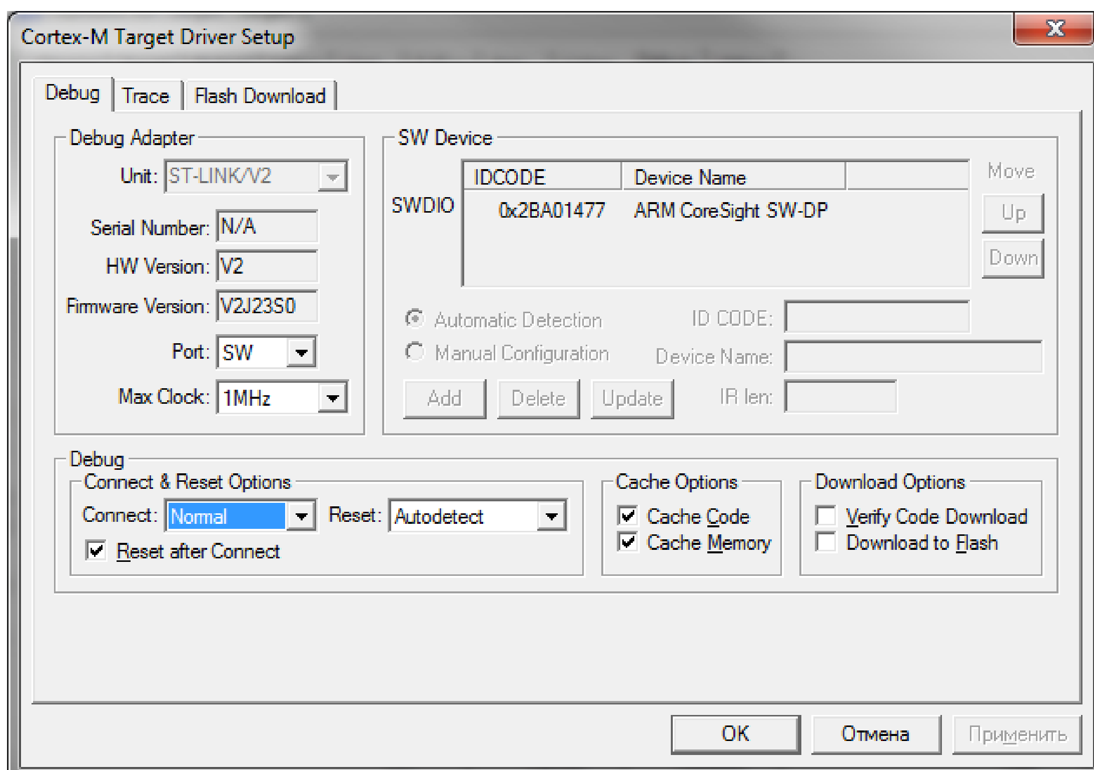

Рис 2.9

Во вкладке Flash Download нажмите add и выберете stm32f4 Flash (рис 2.10). Это действие укажет алгоритм, по которому требуется зашивать данный микроконтроллер. Если используется МК отличный от stm32f4, то нужно выбрать соответствующий ему алгоритм. Выбрав нужный алгоритм, нажмите add.

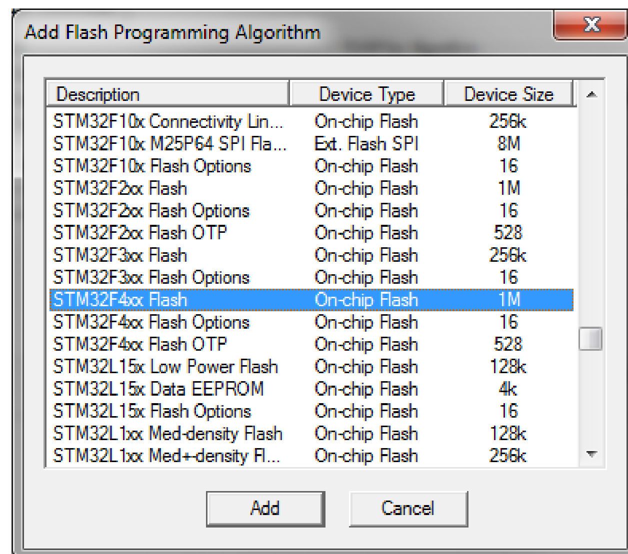

Рис. 2.10

Далее во вкладке Utilities выберете точно такой же программатор. Нажмите Ок. Теперь плата stm32f4-discovery должна запрограммироваться.

При программировании может выскочить окно, предупреждающее об ограничении кода в 32кБ. Это ограничения бесплатной версии программы. Для продолжения необходимо просто нажать Ок.

Использование основных функций компилятора и отладчика.

Компиляция проекта

Для компиляции проекта необходимо нажать на кнопку, отмеченную на рис 2.11.

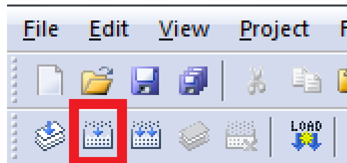

Рис 2.11

При этом должно появиться сообщение сообщающее об отсутствии ошибок (рис 2.12).

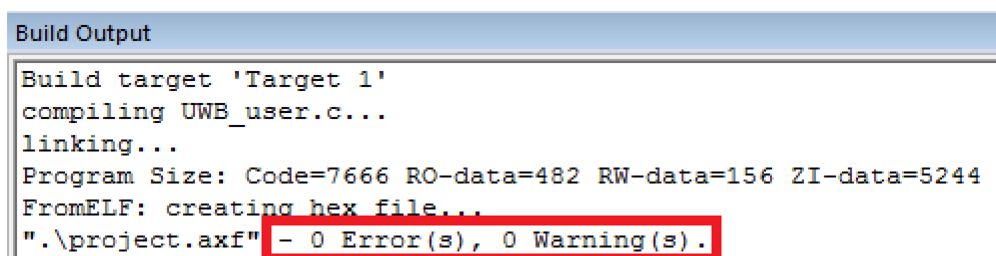

Рис 2.12

Программирование микроконтроллера

Программирование микроконтроллера осуществляется кнопкой, выделенной квадратом на рис. 2.13.

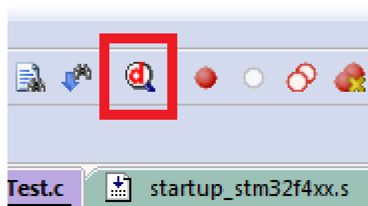

Рис. 2.13

В результате нажатия на эту кнопку программа запишется в микроконтроллер, а сам компилятор войдёт в режим отладки.

Поиск определений функций и переменных

Зачастую в программе появляется необходимость найти описание используемой функции. В компиляторе uVission это делается следующим образом. Щёлкаем по названию правой кнопкой мыши. В контекстном меню выбираем Go to definition. Аналогично можно поступать с переменными.

Использование точек останова

Установить точку останова можно щёлкнув по серому полю правее номера строки. Появится красный кружок как на рис. 2.14 в строке 178.

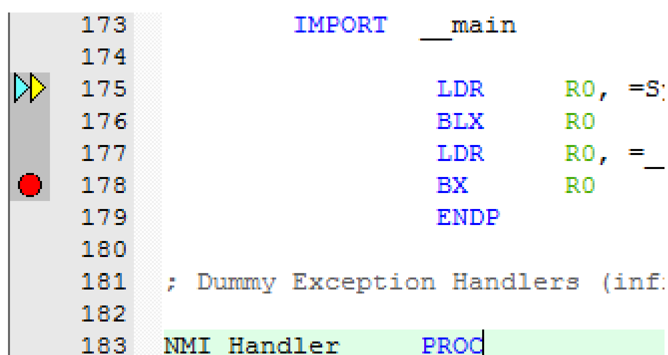

Рис. 2.14

Убрать точку останова можно щелчком по красному кружочку.

Дойдя до точки останова, программа автоматически остановит выполнение на заданной строчке.

На рис. 2.13 можно заметить 2 стрелочки: голубую и жёлтую. Голубая стрелочка показывает на какой строке сейчас находится курсор. А жёлтая – на какой строке остановлена программа.

Выполнение программы по шагам

При остановленном выполнении программы можно двигаться по шагам, нажимая на кнопки, отмеченные на рис 2.15 красным прямоугольником.

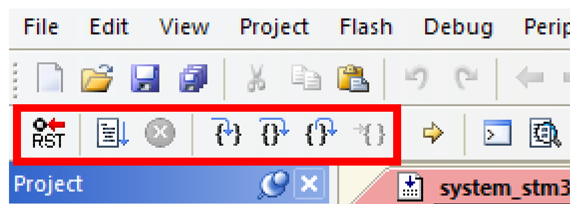

Рис 2.15

Рассмотрим функции этих кнопок слева направо.

1) Сброс микроконтроллера.
2) Запуск программы.
3) Остановка программы
4) 1 шаг с заходом в функцию.
5) 1 шаг без захода в функцию.

При остановленной программе можно посмотреть значение существующей переменной. Для этого можно либо навести на неё курсор мыши или воспользоваться специальным окном внутри компилятора.

Использование окна Watch

Для вызова этого окна нажмите View -> Watch Windows -> Watch 1 (или Watch 2). В открывшимся окошке в самом левом столбце набейте название нужной переменной и нажмите enter.

В открывшемся окне в столбце name необходимо набить название переменной, значение которой требуется узнать и нажать enter. Тогда в разделе value появится значение переменной.

Значение <cannot evaluate> говорит о том, что значение переменной недоступно. Это может быть потому, что переменная локальная и в данной функции она недоступна. Или переменная вообще не существует.

Если необходимо постоянно следить за переменной, лучше всего её сделать глобальной.

Борьба с ошибками компилятора

При написании программ возможно появление различных видов синтактических ошибок или неточностей, которые являются написанными синтаксически правильными, но имею очевидную логическую ошибку, такую как, например, присвоение несогласованных типов данных.

В таблицах 2.1 и 2.2 рассмотрены наиболее типичные ошибки.

При получении любой ошибки или предупреждении можно щёлкнуть по ней дважды левой кнопкой мыши. Программа покажет, в какой конкретно строке была ошибка. Сверившись с таблицей, можно выяснить, что вызвало ошибку и исправить её.

Если программа выдаёт список ошибок, то лучше всего начинать исправление этих ошибок с самой верхней, т.к. зачастую она может являться источником ошибок, появившихся ниже.

На предупреждения так же стоит обращать внимание, т.к. зачастую они указывают на серьёзные ошибки в программе. Например, предупреждение может указать на участки кода, до которых выполнение программы никогда не дойдёт по причине бесконечного цикла над этими участками. Не менее популярной является ошибка с одиночным и двойным равно в if. Это тоже отображается в качестве предупреждения.

Таблица 2.1: Список возможных ошибок

№ ошибки	Название	Описание
5	cannot open source input file "stm32f4xx1.h": No such file or directory	Невозможно открыть заданный файл. Надо проверить папки на его пути. В частности в пути есть ли пробелы в пути к файлу. Они недопустимы.
18	expected a ")"	Пропущена или поставлена лишняя скобочка
20	identifier "GPIO_InitTypeDef" is undefined	Если не определены функции stdperiph, то есть 2 решения: Проверить раскомментирована ли строка #define USE_STDPERIPH_DRIVER в файле stm32f4xx.h Раскомментирована ли соответствующая нужной периферии строка в файле stm32f4xx_conf.h Если это пользовательская функция, то занчит компилятор её почему то не видит. Необходимо проверить наличие этой функции.
35	#error directive: "Please select first …	Не выбран тип МК. В файле stm32f4xx.h раскомментируйте строку #define STM32F40XX
65	expected a ";"	Пропущена точка-запятая.
67	expected a "}"	Пропущена } или поставлена лишняя.
140	too many arguments in function call	Слишком много аргументов было передано функции.
165	too few arguments in function call	Слишком мало аргументов было передано функции.

167	argument of type "int" is incompatible with parameter of type "char *"	Не совместимые типы данных.(указатель и переменная)
268	declaration may not appear after executable statement in block	Описание переменных происходит до первого исполняемого блока.
513	a value of type "char *" cannot be assigned to an entity of type "uint16_t"	Не совместимые типы данных. (указатель и uint16_t).

Таблица 2.2: Список возможных предупреждений

№ ошибки	Название	Описание
12-D	parsing restarts here after previous syntax error	Такая ошибка возникает, например, если описать одну функцию внутри другой.
68-D	integer conversion resulted in a change of sign	Возможная ошибка преобразования типов.
69-D	integer conversion resulted in truncation	Возможная ошибка преобразования типов.
111-D	statement is unreachable	Строка не достижима. Обычно связана с бесконечным циклом перед этой строчкой.
177-D	variable "i" was declared but never referenced	Переменная определена, но не используется.
186-D	pointless comparison of unsigned integer with zero	Может появиться при сравнении без знакового целого с нулём.
187-D	use of "=" where "==" may have been intended	В if написано = вместо ==.

223	function "xxx" declared implicitly	Это предупреждение значит, что какая-то функция используется до определения. Возможно она не определена вообще. Если функция srdperiph, смотрите решение в ошибке 20 г.

Определение частоты тактирования ядра МК

После настройки проекта требуется уточнить тактовую частоту ядра микроконтроллера. Значение частоты тактирования ядра хранится в переменной SystemCoreClock. В начале в неё записано некое значение. Для вычисления значения частоты тактирования ядра используется функция.

SystemCoreClockUpdate();

Эта программа пересчитает тактовую с учётом всех делителей на основе значения HSE_Value, если включено тактирование от внешнего кварца и HSI_Value, если от внутреннего. Если значение HSE_Value выставлено не правильно, то даже при правильном значении SystemCoreClock Мк будет работать не верно.

Программирование портов ввода-вывода.

Настройка портов ввода-вывода с помощью библиотеки CMSIS.

Полным справочником по внутреннему содержанию заданного микроконтроллера является техническая документация на английском языке именуемая Reference manual. В ней описываются общие принципы работы всех периферийных устройств, и за что отвечает каждый управляющий регистр. Если первое является чисто теоретическим и нужно для общего развития и понимания того, что на самом деле происходит, то на втором стоит остановиться подробнее.

Т.к. практически вся настройка периферии – это запись нужных значений в соответствующие биты заданных регистров, то знание какие регистры за что отвечают, является первостепенным для настройки всей периферии.

Возьмем, к примеру, порты ввода-вывода на английском обозначаемые аббревиатурой GPIO (General Port Input Output). За каждый порт отвечает несколько регистров. Их описание дано в конце раздела, посвященному портам ввода-вывода.

Но прежде чем настраивать любую периферию, следует включить её тактирование. Дело в том, что для экономии энергии у всей периферии МК снято тактирование, т.к. она из-за КМОП технологии потребляет энергию только при переключении. Если же нет тактирования, то нет переключений, и через систему ток не протекает, как следствие, энергия не тратится.

За тактирование отвечает система регистров RCC. В технической документации требуется найти какой конкретно регистр отвечает за включение тактирование нужной периферии. Например, если надо включить тактирование порта A, то требуется записать следующую строку:

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
```

Как можно заметить, за тактирование порта A отвечает регистр AHB1ENR. RCC_AHB1ENR_GPIOAEN позволяет установить только

нужный бит, не затрагивая остальных. Для другого порта регистр может отличаться.

Возвращаясь к настройке порта ввода-вывода, рассмотрим, для примера, настройку регистра GPIOx_MODER. Этот регистр отвечает за режим работы порта ввода-вывода. Структура регистра представлена на рис 2.12.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Рис 2.13: Регистр GPIOx_MODER.

Номер после MODER означает номер вывода. Т.е. режим работы программируется двумя битами для каждого вывода.

Могут быть записаны следующие комбинации:

00: Режим входа.

01: Режим выхода.

10: Режим альтернативной функции.

11: Аналоговый режим.

Т.о. записанные «11», к примеру, в биты 27 и 26 настраивают вывод 13 как аналоговый.

Остальные регистры рассматриваются аналогично. Структура и подробное описание этих регистров, как и регистра MODER, находятся в документации Reference manual.

Найти их можно там следующим образом. Открываете содержание (рис 2.15).

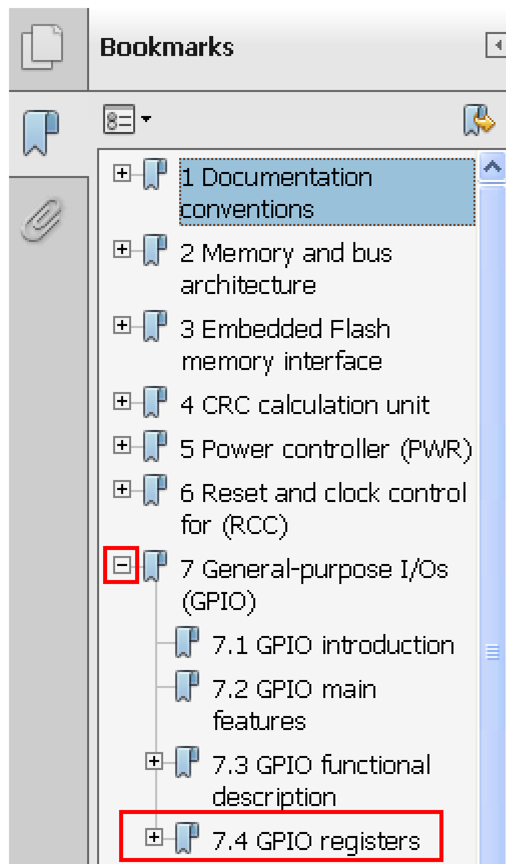

Рис 2.15

Т.к. порты общего назначения называются GPIO, щёлкаете по плюсику напротив соответствующей строки (рис. 2.15) и там выбираете GPIO registers.

Просто перечислим их:

GPIOx_OTYPER отвечает за подтягивание к земле-питанию при работе вывода на выход.

GPIOx_OSPEEDR – отвечает за скорость работы порта при работе его на выход.

GPIOx_PUPDR – отвечает за подтягивание в режиме входа.

GPIOx_IDR – регистр, в который записывает входной значение порта. Работает, если порт сконфигурирован как вход.

GPIOx_ODR - регистр, в который записывается выходное значение порта. Работает, если порт сконфигурирован как выход.

GPIOx_BSRR – Установка 1 в одном младших 16 разрядов, устанавливает на соответствующем выходе 1. Установка 1 в одном старших 16 разрядов, устанавливает на соответствующем выходе 0.

GPIOx_LCKR – Запрещает изменение заданных выводов.

GPIOx_AFRL, GPIOx_AFRH – подключает к выводу альтернативную функцию.

Для программирования портов ввода вывода в библиотеке CMSIS описана специальная структура в файле stm32f4xx.h следующего вида:

```
typedef struct
{
  __IO uint32_t MODER;
  __IO uint32_t OTYPER
  __IO uint32_t OSPEEDR;
  __IO uint32_t PUPDR;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint16_t BSRRL;
  __IO uint16_t BSRRH;
  __IO uint32_t LCKR;
  __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

Программирование уже упомянутого регистра MODER производится следующим образом: GPIOx->MODER = *значение*; где x – название порта.

Заполнять эти регистры можно двумя способами. Первый – это просто записать нужное значение в заданный регистр.

Например, требуется настроить вывод 12 порта B как аналоговый вход. За режим 12-го вывода отвечают биты 24 и 25. Пишем в двоичном коде: 0b00000011000000000000000000000000. Переводим в 16-тиричную форму

записи. Получим: 0x03000000. Т.е. для настройки 12 вывода как аналогового входа надо записать в регистр MODER число 0x03000000. Разумеется, если до этого какие-то другие выводы были настроены ранее, биты их настройки будут перезаписаны. Потому таким способом можно настраивать только сразу все выводы для данного порта, которые используются в системе.

Иногда надо настроить один вывод, не затрагивая остальные. Т.е. нужно установить конкретные биты, не затрагивая уже установленные.

Для установки конкретных бит используются специальные константы, определенные в библиотеке с помощью директивы define. Их названия строятся из названия регистра и номера вывода. Например: GPIO_MODER_MODER0_0 позволяет установить единицу в нулевом бите нулевого вывода следующей строкой:

```
GPIOx->MODER |= GPIO_MODER_MODER0_0;
```

Если надо установить сразу 2 бита одного вывода, к примеру, нулевого, то это делается так:

```
GPIOx->MODER |= GPIO_MODER_MODER0;
```

Если соответствующий бит требуется сбросить, то это делается обратной операцией:

```
GPIOx->MODER &= ~GPIO_MODER_MODER0;
```

Где ~ - побитное НЕ.

Все эти константы описаны в том же файле stm32f4xx.h.

Пример настройки портов ввода-вывода на библиотеке CMSIS

Например, надо настроить нулевой бит порта А как выход. Это будет выглядеть следующим образом:

Первый вариант:

RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;//Включили тактирование.

GPIOA->MODER = 0x00000001;//Установили режим выхода.

GPIOA->OTYPER = 0;//Установили режим Push-Pull

GPIOA->OSPEEDR = 0x00000003;// Установили скорость //100МГц

GPIOA->ODR= 0x000000001;//Зажгли светодиод.

Второй вариант:

RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;//Включили тактирование.

GPIOA->MODER |= GPIO_MODER_MODER0_0;//Установили режим входа.

GPIOA->OTYPER &= ~GPIO_OTYPER_OT_0;//Установили режим Push-Pull

GPIOA->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR0;// Установили скорость //100МГц

GPIOA->ODR|= GPIO_ODR_ODR_0//Зажгли светодиод.

Настройка портов ввода-вывода с помощью библиотеки StdPeriph

В файле stm32f4xx_conf.h находится список всех файлов из библиотеки stdperiph. Комментируя те или иные строки, можно подключать или отключать часть файлов. В нашем случае нам понадобятся только файлы stm32f10x_rcc.h и stm32f10x_gpio.h, отвечающие за тактирование и порты ввода-вывода соответственно. А также misc.h.

Вместе с библиотекой Std_periph поставляется справочный материал по работе с ней. Из-за плохой структурированности данного материала его практическое применение затруднено. Поэтому проще изучить возможности библиотеки, анализируя ее исходный код, в котором присутствуют развернутые комментарии к каждому его участку. Рассмотрим программирование портов микроконтроллера с помощью stdPeriph.

Как уже было отмечено, прежде всего надо включить тактирование порта. За тактирование всей периферии отвечает раздел RCC и все процедуры, отвечающие за тактирование микроконтроллера расположены в файлах stm32f4xx_rcc.h и stm32f4xx_rcc.c. В файле stm32f4xx_rcc.h находим define, в котором используется название порта, например GPIOC. Это RCC_AHB1Periph_GPIOC. А значит, для включения тактирования нужна функция, воздействующая на регистр AHB1. Эта функция RCC_AHB1PeriphClockCmd. В итоге получаем следующее:

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
```

За функции связанные с портами ввода-вывода отвечают файлы stm32f4xx_gpio.h и stm32f4xx_gpio.c. Для инициализации порта надо вызвать функцию GPIO_Init. Эта функция принимает 2 параметра. Первый – это название порта, второй – структура GPIO_InitTypeDef с настройками.

Вид этой структуры описан в файле stm32f10x_gpio.h:

```
typedef struct
{
```

uint16_t GPIO_Pin; /* Описывает какие выводы порта будут инициализированы. Этот параметр может принимать одно из значений, описанных в структуре GPIO_pins_define */

GPIOSpeed_TypeDef GPIO_Speed; /* Устанавливает скорость работы вывода. Этот параметр может принимать значения из структуры GPIOSpeed_TypeDef */

GPIOMode_TypeDef GPIO_Mode; /* Определяет режим работы выбранных выводов МК. Этот параметр может принимать значения из структуры GPIOMode_TypeDef */

GPIOOType_TypeDef GPIO_OType; /* Определяет режим работы выводов, если те проинициализированы как выход. Этот параметр может принимать значения из структуры GPIOOType_TypeDef*/

GPIOPuPd_TypeDef GPIO_PuPd; /* Определяет подтягивание вывода к земле или питанию. Этот параметр может принимать значения из структуры GPIOPuPd_TypeDef*/

}GPIO_InitTypeDef;

GPIO_Pin может принимать значения вида: GPIO_Pin_x, где x – номер вывода в порте.

GPIO_Speed может принимать одно из четырёх значений:

GPIO_Speed_2MHz,

GPIO_Speed_25MHz,

GPIO_Speed_50MHz,

GPIO_Speed_100MHz.

Работает только, если вывод определён как выход.

GPIO_Mode принимает следующие значения:

Таблица 2.3

Название	Описание
GPIO_Mode_IN	режим входа
GPIO_Mode_OUT	режим выхода

GPIO_Mode_AF	режим альтернативной функции
GPIO_Mode_AN	режим аналогового входа

GPIO_OType принимает значения. Работает только если вывод определён как выход:

Таблица 2.4

Название	Описание
GPIO_OType_PP	есть подтягивание
GPIO_OType_OD	открытый коллектор

GPIO_PuPd принимает значения. Работает только если вывод определён как вход:

Таблица 2.5

Название	Описание
GPIO_PuPd_NOPULL	без подтягивания
GPIO_PuPd_UP	подтягивание к питанию
GPIO_PuPd_DOWN	подтягивание к земле

При программировании вывода микроконтроллера заполняется описанная выше структура и вызывается функция GPIO_Init.

Теперь заполним структуру нужными нам параметрами.

```
GPIO_InitTypeDef GPIO_InitType;

GPIO_InitType.GPIO_Pin = GPIO_Pin_9;
GPIO_InitType.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitType.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitType. GPIO_OType = GPIO_OType_PP;
GPIO_InitType. GPIO_PuPd = GPIO_PuPd_NOPULL;
```

В случае, если необходимо инициализировать одинаково разные выводы одного порта, номера этих выводов пишутся через знак побитового ИЛИ. Например:

GPIO_InitType.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_10;

3) Теперь запишем настройки в порт:

GPIO_Init(GPIOA, & GPIO_InitType);

Первый параметр – это название порта в виде GPIOX, где X - номер порта. Второй параметр – указатель на структуру с настройками.

Для установки на вывод высокого уровня напряжения (“1”) используется функция:

GPIO_SetBits(GPIOB, GPIO_Pin_0);

Где GPIOB – название порта, в GPIO_Pin_x, где «x» – это номер вывода.

Для установки на вывод низкого уровня напряжения (“0”) используется функция:

GPIO_ResetBits(GPIOB, GPIO_Pin_1);

Пример настройки портов ввода-вывода с помощью библиотеки StdPeriph

Предположим, что надо настроить вывод B12 как выход. Тогда надо написать следующие строки:

```
GPIO_InitTypeDef     GPIO_InitType;
//Включаем тактирование
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
// Заполняем структуру
GPIO_InitType.GPIO_Pin = GPIO_Pin_12;
GPIO_InitType.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitType.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitType. GPIO_OType = GPIO_OType_PP;
GPIO_InitType. GPIO_PuPd = GPIO_PuPd_NOPULL;
// Записываем структуру в микроконтроллер
GPIO_Init(GPIOB, & GPIO_InitType);
```

Программирование таймера

Настройка таймера общего назначения

Микроконтроллер stm32f4 содержит в себе 14 таймеров. Каждый таймер обладает какой-то особенностью, которой может не быть у других. Некоторые таймеры могут считать вверх и вниз, другие только вверх. Есть 32-битные таймеры, есть 16-битные. Некоторые таймеры могут отправлять запросы на ДМА. Поэтому перед использованием таймера требуется подобрать наиболее подходящий под решаемую задачу.

Таблица 2.6

№	Разрядность	Режимы работы	Внешние входы
1,8	16 бит	Вверх, вниз, вверх-вниз	4
2,5	32 бит	Вверх, вниз, вверх-вниз	4
3,4	16 бит	Вверх, вниз, вверх-вниз	4
9	16 бит	Вверх	2
10,11	16 бит	Вверх	1
12	16 бит	Вверх	2
13,14	16 бит	Вверх	1
6,7	16 бит	Вверх	0

Таймеры 6,7 являются базовыми с наименьшим набором функций.

Таймеры 1 и 8 являются наиболее сложными и многофункциональными.

Остальные таймеры представляют собой нечто среднее и именно они обычно применяются для решения возникающих задач.

Рассмотрим настройку таймера на примере таймера 2.

Прежде всего необходимо включить тактирование:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
```

Затем заполнить структуру настроек. Структура включает в себя параметры, представленные в табл.2.7.

Табл. 2.7

Название	Описание
TIM_Prescaler	Предделитель тактовой частоты. Может изменяться от 0 до 65535. Микроконтроллер к записанному значению добавляет 1. Т.о. запись в данный параметр нуля означает деление на 1.
TIM_CounterMode	Режим направления счёта счётчика (Вверх, вниз, вверх-вниз)
TIM_Period	Этот параметр задаёт значение на котором будет переполнение таймера.
TIM_ClockDivision	Настраивает внешний делитель.
TIM_RepetitionCounter	Устанавливает число повторов. Когда это число становится равным 0 генерируется событие переполнения. Так, например, можно задавать число ШИМ периодов.

Разумеется, при настройке таймера следует заранее узнать частоту тактирования этого таймера. От неё напрямую зависят необходимые настройки. Как узнать тактовую частоту сигнала, подаваемого на таймер, описано в разделе «Как узнать частоту тактирования таймера».

Записываем в микроконтроллер:

TIM_TimeBaseInit(TIM2, &TIM_Time_user);

Теперь осталось только разрешить глобальные прерывания:

NVIC_EnableIRQ(TIM2_IRQn);

Прерывания по переполнению:

TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

И включить таймер:

```
TIM_Cmd(TIM2, ENABLE);
```

Теперь каждые 20 мс программа будет попадать в функцию:

```
void TIM2_IRQHandler(void)
{
}
```

В данной функции требуется проверять установлен ли флаг прерывания по переполнению и сбрасывать его. В итоге функция примет вид:

Пример инициализации таймера общего назначения

Например, пусть частота тактирования таймера равна 10МГц. Требуется настроить таймер с переполнением каждые 20 мс.

T=20 мс => f=1/20 мс = 0.05 кГц = 50 Гц. – частота переполнений таймера.

Пусть делитель будет равен 1 000. Тогда частота увеличения счётчика на 1 равна 10 кГц. Чтобы получить 50 Гц, требуется поделить частоту ещё на 200. Т.е. при частоте тактирования в 10 кГц за 20 мс таймер насчитает 200 импульсов.

Получаем следующие настройки:

```
TIM_TimeBaseInitTypeDef TIM_Time_user;

TIM_Time_user.TIM_Prescaler = 1000-1;
TIM_Time_user.TIM_CounterMode = TIM_CounterMode_Up;
TIM_Time_user.TIM_Period = 200;
TIM_Time_user. TIM_ClockDivision = TIM_CKD_DIV1;

TIM_TimeBaseInit(TIM2, &TIM_Time_user);

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

```
//Функция прерывания от таймера
void TIM2_IRQHandler(void)
{
  if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
  {
          TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
  }
}
```

После строки TIM_ClearITPendingBit можно описывать то, что должно происходить каждые 20 мс.

Инициализация системного таймера

Задержку программы можно задавать в тактах, но такую задержку сложно пересчитать в реальное время. Привязку к реальному времени можно сделать с помощью упомянутого ранее таймера реального времени.

Инициализация таймера производится следующей строкой:

```
SysTick_Config(SystemCoreClock /x);//
```

Число x говорит сколько раз за секунду будет срабатывать прерывание от этого таймера.

В переменной SystemCoreClock записана реальная частота тактирования микроконтроллера. Она может отличаться от HSE, т.к. в микроконтроллере есть так называемая PLL, которая может умножать и делить тактовую частоту в заданное число раз. Чтобы узнать текущее реальное значение тактовой частоты микроконтроллера используется функция:

```
SystemCoreClockUpdate();
```

Рекомендуется использовать её до инициализации системного таймера.

При срабатывании прерывания от SysTick, будет вызываться функция:

```
void SysTick_Handler(void);
```

В эту функцию надо писать то, что должно произойти в момент возникновения прерывания.

Пример настройки системного таймера

```
void SysTick_Handler(void)
{


}


void main(void)
{
  SysTick_Config(SystemCoreClock/1000);//1ms
}
```

К примеру, рассмотрим алгоритм реализации функции задержки с помощью этого таймера:

Наша функция должна принять входной параметр, говорящий, на сколько миллисекунд должна быть произведена задержка. Функция первым делом копирует значение в глобальную переменную, которую системный таймер уменьшает каждый цикл прерывания. Теперь функции остаётся только подождать пока переменная не станет равной нулю. После чего завершить свою работу.

Алгоритм реализации длительных временных задержек

Зачастую возникает потребность в организации длительных задержек, кратных времени срабатывания системного таймера. Для организации таких задержек создаётся переменная, в которую будет записываться время задержки. При этом в прерывании системного таймера записывается код:

```
if (a>0)
{
  a--;
}
```

Когда же переменная a достигает значения, равного 1, то выставляется флаг. Если выставлять флаг при равенстве a нулю, то он будет выставляться

каждое срабатывание прерывания системного таймера, пока в а не будет записано значение отличное от 0.

В итоге в функции прерывания имеем код вида:

```
if (a>0)
{
  a--;
  if (a == 1)
  {
          Flag=1;
  }
}
```

В самой программе проверяется флаг и по его установке выполняется какое-либо действие. Например инвертируется состояние светодиода

```
while(1)
{
      if (Flag==1)
      {
          Flag = 0;
  GPIO_ToggleBits(GPIOA, GPIO_Pin_8);
      }
}
```

Светодиод поменяет своё состояние после установки флага, как только программа дойдёт до этого места. Очевидно, что это может быть не сразу, как только установится флаг. Если же нужна мгновенная реакция, то инверсию необходимо прописать прямо в функции прерывания. Но стоит помнить, что при длительном выполнении функции прерывания есть шанс пропустить другое прерывание.

Либо можно аналогичный код писать прямо в функции прерывания. Например, если требуется мигать светодиодом, то можно в функции прерывания написать так:

```
if (i > 0)
{
        i--
} else {
        GPIO_ToggleBits(GPIOD, GPIO_Pin_12);
        i=1000;
}
```

Как только произойдёт 1000 прерываний, переменная i обнулится и состояние GPIOD.12 поменяется. Далее цикл начнётся заново.

Как узнать частоту тактирования таймера

Для того, чтобы узнать частоты тактирующих сигналов различных шин микроконтроллера возможно воспользоваться специальной функцией:

```
void    RCC_GetClocksFreq(RCC_ClocksTypeDef* RCC_Clocks);
```

В качестве её аргумента используется структура, в которую записываются значения тактовых сигналов различных шин. Их значения приведены в табл.2.8.

Таблица 2.8

Название	Описание
SYSCLK_Frequency	Системная частота тактирования
HCLK_Frequency	Частота тактирования ядра
PCLK1_Frequency	Частота тактирования шины PCLK1, отвечающей за тактирование таймеров 2-7, 12-14.
PCLK2_Frequency	Частота тактирования шины PCLK1, отвечающей за тактирование таймеров 1, 8-11.

Частота тактирования таймера может отличаться от частоты тактирования шины в 1 или 2 раза. Если HCLK_Frequency и PCLKx_Frequency совпадают, то частота тактирования соответствующих таймеров равна PCLK1_Frequency. Если PCLK1_Frequency меньше HCLK_Frequency, то частота таймера в 2 раза больше частоты PCLKx_Frequency.

Алгоритмы борьбы с дребезгом

Общий обзор алгоритмов борьбы с дребезгом

Наиболее популярны 2 алгоритма борьбы с дребезгом. Первый – это реакция на первый же появившийся импульс и ожидание после этого времени, через которое дребезг гарантированно пройдёт.

Вторым является алгоритм с набором статистики. Этот алгоритм заключается в том, что если на входе устройство превалирует 0, то делается вывод, что на входе 0. И наоборот.

Разумеется борьба с дребезгом нужна только, если требуется поймать момент нажатия на кнопку или отжатия оной.

Рассмотрим оба варианта отдельно.

Алгоритм борьбы с дребезгом с помощью задержки

В этом варианте проверяется, нажата ли кнопка. Если кнопка нажата, то совершается действие и проверка кнопки отключается на заданное время. Отключение можно производить с помощью функции delay или с помощью флага.

Алгоритм программы для борьбы с дребезгом с помощью функции delay может выглядеть примерно так, как представлено на рис 2.16.

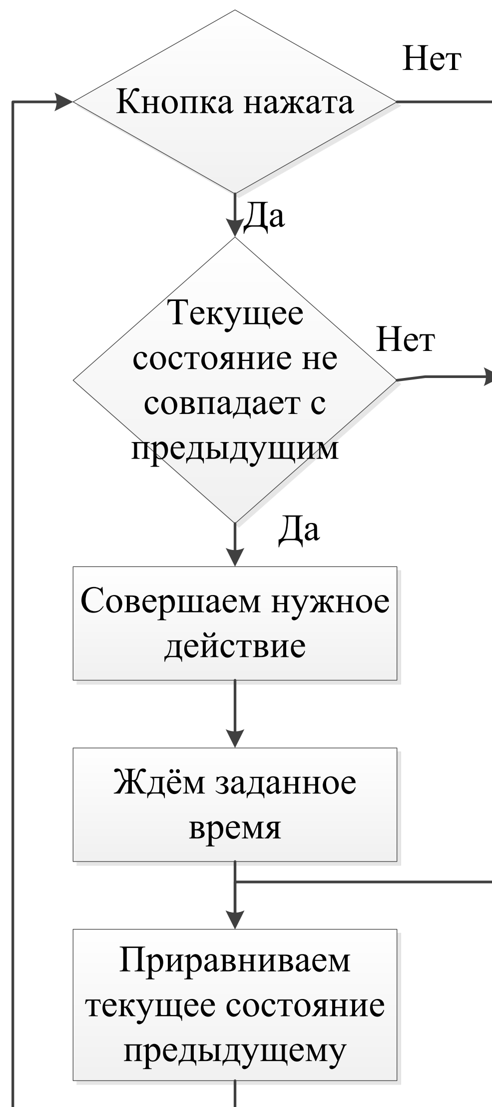

Рис 2.16

На рис 2.16 проверяем, нажата ли кнопка. Если кнопка нажата, то проверяем, совпадает ли текущее состояние кнопки с предыдущим. Т.к. кнопка только что была нажата, то её состояние не совпадает и условие выполняется. Далее совершаем нужные нам действия, которые должны быть совершены по нажатию на кнопку и ждём некоторое время, пока дребезг гарантированно не пропадёт. В конце присваиваем предыдущему состоянию следующее.

Если при следующем выполнении цикла, кнопка ещё будет нажата, текущее состояние уже будет равно предыдущему и условие не выполнится. Т.о. отлавливается только нажатие на кнопку.

Если же ожидать по флагу, то алгоритм немного поменяется на тот, что показан на рис 2.17.

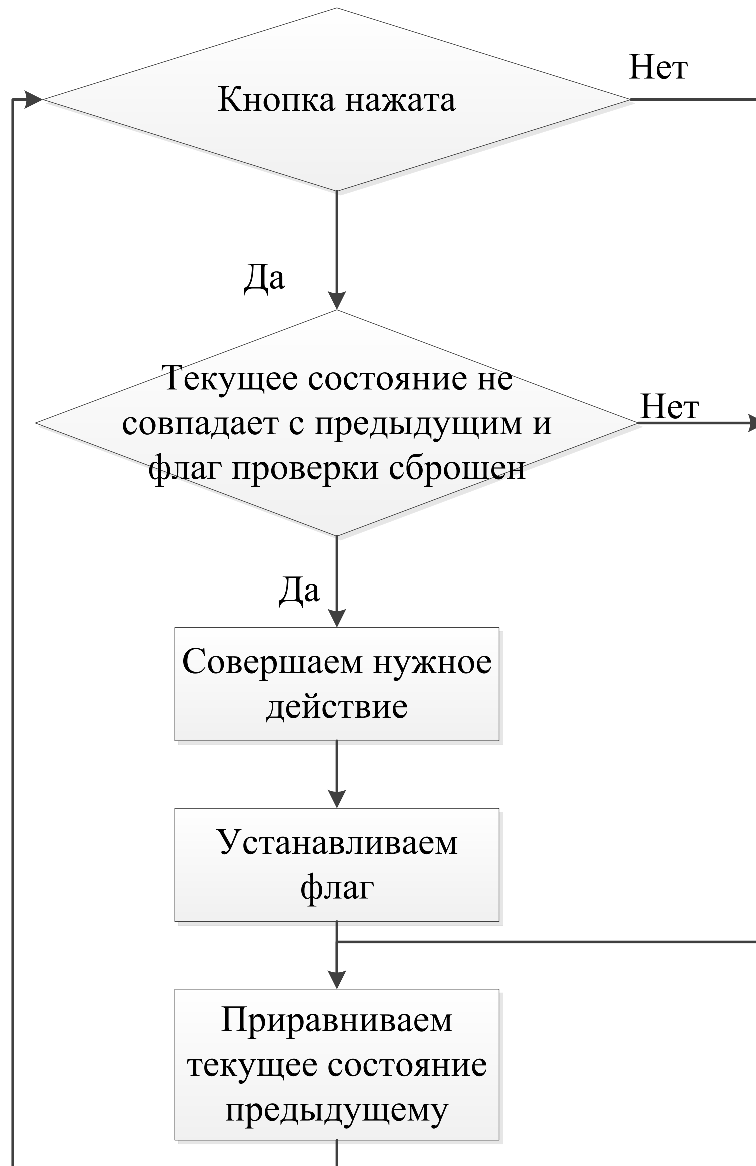

Рис 2.17

Отличием алгоритма на рис 2.16 от алгоритма на 2.17 является то, что при нажатии кнопки устанавливается флаг, запрещающий проверку кнопки. Сам флаг через заданное время сбрасывается в системном таймере.

Гистерезисный алгоритм борьбы с дребезгом

Алгоритмов, реализующих данный метод много. Рассмотрим один из них. Суть его заключается в получении интегральной составляющей входного сигнала. Т.е. какой сигнал чаще встречается (ноль или единица), в пользу того сигнала и делается вывод. Алгоритм данного анализа приведён на рис 2.18.

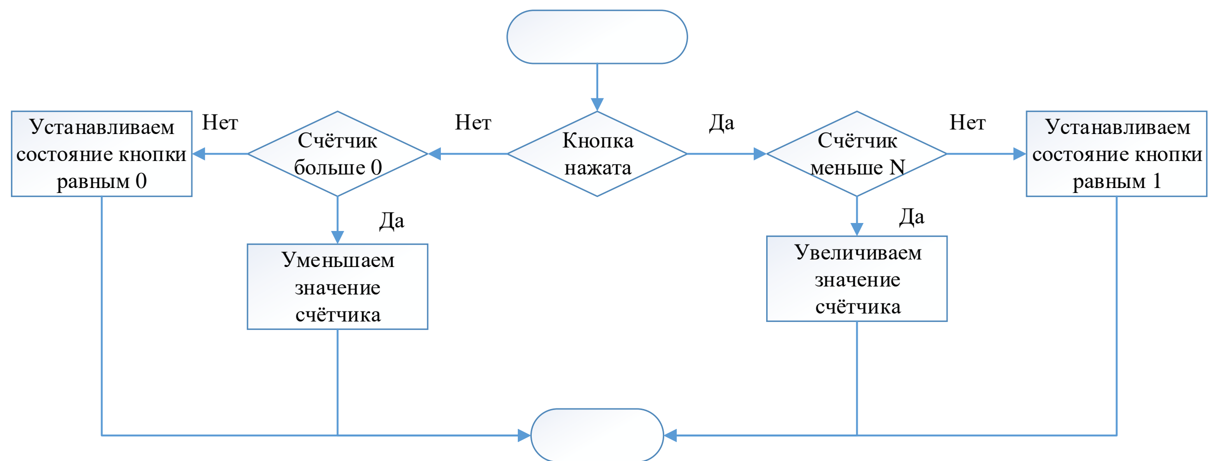

Рис 2.18

Алгоритм на рис 2.18 вызывается периодически в заданное время и проверяет, нажата ли кнопка. Если она нажата, то переменная Count увеличивается на 1, если отжата, то уменьшается. В результате, значения N или нуля она достигнет только тогда, когда ноль или единица явно преобладают в системе.

Такой способ обработки сигнала напоминает петлю гистерезиса, в которой обратный ход характеристики отличается от прямого.

Изменением значения N можно менять чувствительность кнопки, т.к. при больших значениях N могут пропускаться нажатия, а при очень маленьких – проскакивать дребезг.

Выход данного алгоритма уже можно рассматривать как кнопку без дребезга. Т.е. использовать алгоритмы, рассмотренные выше, но без функции задержки, т.к кнопка – идеальная.

Настройка USART с помощью библиотеки StdPeriph.

Описание общего алгоритма настройки

Программирование USART в STM32 происходит по следующему алгоритму:

0) Подключить файл, отвечающий за программирование USART в StdPeriph.

1) Включаем тактирование порта (портов), на который выведен USART.

2) Инициализируем выводы USART на работу в альтернативном режиме.

3) Включаем тактирование USART.

4) Инициализируем USART.

5) Разрешаем глобальные прерывания прерывания.

6) Разрешаем прерывания от USART.

7) Включаем работу USART.

На этом инициализация заканчивается.

Пройдёмся по всему по порядку.

0) Для подключения USART надо расскоментировать строку #include "stm32f4xx_usart.h" в файле **stm32f4xx_conf.h.**

1) Включение тактирования порта смотрите в лабораторной 1.

2) Stm32f4xx имеет в наличии 3 USART. Каждый вывод USART-а может быть подключен к одному из двух заранее заданных выводов. Конкретные номера выводов можно узнать из технической документации на конкретный микроконтроллер, в которой описываются значения всех выходов.

Подключение выводов к USART производится функцией GPIO_PinAFConfig.

Например, так:

GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_USART3);

Первым параметром указывается порт на котором необходимый находится вывод. Вторым – номер вывода. Эти номера описаны в виде

GPIO_PinSourcex, где x – номер вывода. Вместо этой константы можно прямую записывать номер вывода. Третьим параметром указывается, что именно требуется подключить к данному выводу. Разумеется, это подключение должно быть осуществимо физически. Т.е. требуется заранее убедиться, что к данному выводу можно подключить данное периферийное устройство.

При настройке выводов выбирается режим альтернативной функции. Чтобы USART мог успешно управлять выводом, используемым для передачи данных, его следует настроить на подтягивание к питанию-земле.

Для правильной работы входного вывода USART, его требуется подтянуть к питанию, т.к. пассивным состоянием USART считается «1».

Скорость обычно выбирают равной 50МГц.

3) Включение тактирования USART производится функцией:

RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

Где первый параметр – регистр RCC, второй – команда.

Список устройств, которые включаются этой функцией можно найти в файле *stm32f10x_rcc.h.*

4) Собственно сама инициализация:

Структура инициализирующая USART описана в файле *stm32f4xx_usart.h* и обозначается как USART_InitTypeDef.

Примечание*: Прежде всего, разберемся, как же заполнять подобные структуры. Сама структура описана, как уже было сказано в файле stm32f4xx_usart.h. Рядом с каждым параметром этой структуры дано описание, для чего нужен тот или иной параметр и как он инициализируется. Если для инициализации какого-то параметра нужно использовать определённые ниже define, то в комментариях к параметру указывается ссылка на этот define начинающаяся ключевым словом @ref. Далее следует слово, набив которое в поиск, можно без труда отыскать нужные define. Например, параметр USART_StopBits. У него в комментариях указана строка: @ref USART_Stop_Bits. Скопировав слово USART_Stop_Bits в*

поиск можно найти, что данный параметр принимает одно из 4-х значений: USART_StopBits_1, USART_StopBits_0_5, USART_StopBits_2, USART_StopBits_1_5. Последний define проверяет, является ли указанный define элементом USART_Stop_Bits. Для инициализации он не используется.

Рассмотрим элементы этой структуры:

В *USART_BaudRate* описывается скорость передачи и приёма данных. Микроконтроллер может выставить практически любую скорость передачи. Это значение ограничено только частотой тактирования. Причём ограничения касаются не только максимального диапазона, но и точности тактирования USART. Обычно скорость передачи USART принимает одно из следующих стандартных значений: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600. По-умолчанию это значение обычно принимается равным 9600. Задаётся в численном виде.

USART_WordLength задаёт длину одной посылки. Stm32f4xx поддерживает только 2 длины в 8 и 9 бит, которые обозначаются соответственно USART_WordLength_8b и USART_WordLength_9b. По умолчанию это значение обычно принимается равным 8 бит.

USART_StopBits – задаёт количество стоп бит. По умолчанию количество стоп-бит равно 1.

USART_Parity – проверка на чётность для увеличения вероятности правильной передачи данных. По умолчанию проверка обычно отключается.

USART_Mode определяет режим работы USART. Будет ли тот работать только на передачу, только на приём или сразу в обе стороны. Обозначается соответственно USART_Mode_Tx, USART_Mode_Rx и USART_Mode_Tx|USART_Mode_Rx.

USART_HardwareFlowControl включает аппаратное управление потоком. Может принимать одно из следующих значений:

USART_HardwareFlowControl_None

USART_HardwareFlowControl_RTS

USART_HardwareFlowControl_CTS

USART_HardwareFlowControl_RTS_CTS

По-умолчанию управление потоком обычно отключается.

Инициализация производится функцией USART_Init. Например:

USART_Init(USART1, &USART_InitStructure);

Где первый параметр – идентификатор USART. Второй – Указатель на заполненную структуру.

5) Включение прерываний от USART

NVIC_EnableIRQ(USART1_IRQn);

6) разрешаем прерывания по приёму:

USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);

7) Включаем USART.

USART_Cmd(USART1, ENABLE);

Теперь опишем функцию прерывания от USART, которая возвращает принятый байт. Только тут стоит учитывать 2 момента:

1) Флаг прерывания зачастую аппаратно не сбрасывается.

2) У USART все прерывания вызывают одну и ту же функцию.

Т.е. нам надо различать прерывания от разных источников и программно сбрасывать флаг прерывания. В итоге функция, вызываемая USART выглядит, так:

```
void USART1_IRQHandler(void)
{
uint8_t Res_data=0;
// Обработка события RXNE (приёма)
if ( USART_GetITStatus(USART1, USART_IT_RXNE) )
{
// очистка бита прерывания
USART_ClearITPendingBit(USART1, USART_IT_RXNE);
//Сюда пишется, что должно произойти приёме одного байта
Res_data = USART_ReceiveData(USART1);//Приняли байт
```

```
send_to_USART(Res_data);//Отправили обратно
};

}
```

И последнее, перед посылкой ожидайте отправки предыдущего байта с помощью строки:

```
while (!USART_GetFlagStatus(USART1, USART_SR_TXE)) {}
```

Т.е. функцию отправки одного байта данных можно написать следующим образом:

```
//--------------------------------------------------------------------------------
//Функция отправляющая байт в USART
//--------------------------------------------------------------------------------
void send_to_USART(uint8_t data)
{
    while(!(USART1->SR & USART_SR_TC)); //Ждем пока бит TC в регистре SR станет 1
    USART1->DR=data; //Отсылаем байт через USART
}
```

Аналогично для передачи строки данных можно использовать процедуру:

```
void SendString_to_ USART (const char *str)
{
  while(*str != '\0')
  {
    while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
    USART_SendData(USART1, *str++);
  }
}
```

Пример настройки USART

Рассмотрим пример в котором нам надо настроить USART на выход на скорости 4800 бит в секунду.

GPIO_InitTypeDef GPIO_USART_ini; //Структура для инициализации выводов

USART_InitTypeDef USART1_User_ini; //Структура для инициализации USART

//Включение тактирования вывода по которому будет передавать данные USART

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

//Подключение вывода к USART. Последняя цифра – номер USART.

GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_USART3);//

GPIO_USART_ini.GPIO_Pin = GPIO_Pin_14;

GPIO_USART_ini.GPIO_Mode = GPIO_Mode_AF;

GPIO_USART_ini.GPIO_Speed = GPIO_Speed_50MHz;

GPIO_USART_ini.GPIO_OType = GPIO_OType_PP;

GPIO_USART_ini.GPIO_PuPd = GPIO_PuPd_UP;

GPIO_Init(GPIOB, &GPIO_USART_ini);

USART1_User_ini.USART_BaudRate = 4800;

USART1_User_ini.USART_WordLength = USART_WordLength_8b;

USART1_User_ini.USART_StopBits = USART_StopBits_1;

USART1_User_ini.USART_Parity = USART_Parity_No;

USART1_User_ini.USART_HardwareFlowControl = USART_HardwareFlowControl_RTS_CTS;

USART1_User_ini.USART_Mode = USART_Mode_Tx;

USART_Init(USART3, &USART1_User_ini);

NVIC_EnableIRQ(USART3_IRQn);//Включаем общие прерывания USART.

USART_Cmd(USART3, ENABLE);//Включаем USART

USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);//Настраиваем прерывания на приём (если есть).

Протоколы

Протокол передачи данных – это набор правил, по которым общаются 2 и более устройств. Этот набор включает в себя порядок и тип передаваемых данных, алгоритм генерации и приёма пакетов, алгоритм распознавания своего пакета и т.д.

Протоколы могу быть как очень простыми, в которых данные передаются 1 байтом, так и с очень сложной многоуровневой структурой.

В протоколе обычно определяются формат пакета с данными, формат данных внутри пакета, условия передачи тех или иных пакетов. Например, пакеты могут передаваться через заданные промежутки времени. Такими пакетами могут быть пакеты, опредляющие наличие подключения.

Формат кодирования данных

Данные в пакетах можно кодировать различными способами. Самый очевидный способ записывать числа так, как они представлен на ПК. Т.е., если надо передать число 100, можно просто передать один байт с числом 100 в формате беззнакового целого. На приёме такое число можно сразу применять для дальнейшей работы. Такой способ кодирования можно назвать *бинарным*.

Этот протокол позволяет создавать данные с пакетом наименьшего размера. И, если данные передаются между двумя устройствами, позволяет наиболее просто принимать данные из полученного пакета.

Но такой способ передачи данных имеет ряд недостатков. При отображении такие пакеты сложно воспринимать человеку. Например, если требуется передать число более 256, необходимо использовать 2 байта и при визуализации в уме или отдельной программе собирать эти байты в одно число. Также сложно запретить какие-либо комбинации бит в байте, чтобы использовать эти комбинации как служебные.

Эти проблемы решает *символьный* формат кодирования данных. В этом случае все числа передаются символами. Т.е., если надо передать число 100, передаётся символ ‘1’, потом символы '0' и ‘0’. В бинарном коде получается такой набор данных: 0x313030. Плюсом такого кодирования данных является лёгкость восприятия человеком и возможность использовать служебные комбинации. Например, 0 обычно используется как стоп байт. Если при таком способе кодирования данных, был принят 0, то считается, что пакет принят полностью.

Одновременно с этим, можно просто проверить пакет на явные ошибки без использовать CRC. Для этого просто проверяется, что в пакете нет неиспользуемых символов.

Примеры наиболее часто используемых полей пакетов

Рассмотрим различные структуры пакетов. Какие поля встречаются в пакетах и для чего они используются. Знание стандартных полей и их назначения позволяет проще определяться с форматом пакета собственного протокола.

Поле команд

Самый простой протокол включает в себя только поле команд. По каждой команде МК выполняет некие действия. Пакеты, состоящие только из поля команд можно использовать, если не требуется передавать данные.

Простейшим случаем является однобайтная передача данных, в которое команда представляет собой данные размером в 1 байт. Этот байт передаёт всю необходимую информацию. Количество различных передач равно 256, чего хватает для решения примитивных задач по общению.

Данный протокол не подразумевает наличия стартовых и стоповых последовательностей. Реализовать его можно достаточно просто:

```
void USART1_IRQHandler(void)
{
uint8_t Res_data=0;
// Обработка события RXNE (приёма)
if ( USART_GetITStatus(USART1, USART_IT_RXNE) )
{
// очистка бита прерывания
USART_ClearITPendingBit(USART1, USART_IT_RXNE);
//Сюда пишется, что должно произойти приёме одного байта
Res_data = USART_ReceiveData(USART1);//Приняли байт
if (Res_data == 0x01)//Если принятый байт равен 1, …
{
  RED_ON();//… зажигаем светодиод.
}
}
}
```

При более сложных задачах, выполняемых достаточно долго. Рекомендуется их выносить в функцию main. Для этого при приёме байта выставляется флаг окончания приема, и этот флаг проверяется уже в функции main. Если флаг установлен, производится обработка принятого байта.

```
uint8_t receive_flag=0;
uint8_t Res_data=0;

void USART1_IRQHandler(void)
```

```
{

  // Обработка события RXNE (приёма)
  if ( USART_GetITStatus(USART1, USART_IT_RXNE) )
  {
  // очистка бита прерывания
  USART_ClearITPendingBit(USART1, USART_IT_RXNE);
  //Сюда пишется, что должно произойти приёме одного байта
  Res_data = USART_ReceiveData(USART1);//Приняли байт
  receive_flag = 1;
  }
}

int main()
{
   if (receive_flag == 1)
  {
          receive_flag = 0;//Сбрасываем флаг.
          switch (Res_data){
  case 0x01:{RED_ON();}break;
  case 0x02:{RED_OFF();}break;
  }
   }
}
```

Но возможен вариант, когда обработка будет настолько долгая, что успеет прийти следующий байт. Но тогда предыдущий будет затёрт. Чтобы этого не случилось, создаётся буфер, представляющий из себя массив данных типа char. Определяемый, к примеру, следующим образом:

```
uint8_t receive_buffer[256];
```

Для работы с этим буфером нам требуется ряд переменных:

1) Указатель (не путать с указателями в Си) на место записи принятых данных.

2) Указатель на место чтения обрабатываемых данных.

3) Количество необработанных данных в буфере.

```
uint8_t receive_write=0, receive_read=0, receive_count=0;
```

Заполнение буфера тогда выглядит так:

```
receive_buffer[receive_write] = USART_ReceiveData(USART1);//Приняли байт
receive_write++;
receive_count++;
```

О переполнении можно не беспокоиться благодаря выбранному размеру буфера. По достижении 255 все переменные будут обнуляться автоматически.

Тогда, обработка будет выглядеть так:

```
if (receive_count > 0)
{
  switch (receive_buffer[receive_read]){
case 0x01:{RED_ON();}break;
case 0x02:{RED_OFF();}break;
}
receive_count--;
receive_read++;
}
```

Поле данных

Зачастую 256 значений недостаточно. Либо команда требуется пояснения в виде дополнительных данных. Тогда в пакета вводится поле данных . Оно может быть либо постоянной длины, либо переменной. Для простоты оно делается постоянной длины для каждой команды.

На рис. 2.19 приведён пример пакета с командой и данными для них.

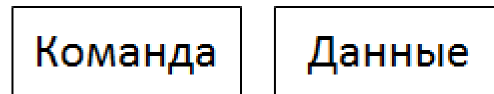

Рис. 2.19

В команде отправляется то что это за данные, а в данных – нужная информация.

Например, можно отправлять команду светодиод, а в поле данных номера светодиодов, которые надо включить.

При приёме поля данных можно отсчитывать сколько байт пришло. Например, алгоритм обработки может быть изменён так:

```
uint8_t command, data, red_delay;

if (receive_count > 1)
{
        command = receive_buffer[receive_read];
        receive_read++;
        data = receive_buffer[receive_read];
 receive_read++;

 switch (command){
 case 0x01:{RED_ON();}break;
 case 0x02:{RED_OFF();}break;
 case 0x03:{red_delay = data;}break;
 }
 receive_count = 0;
```

```
}
```

Где red_delay – это некая переменная, показывающая сколько времени горит или не горит красный светодиод.

Это рабочий вариант, но не надёжный, т.к. не учитывает вариант, что данные могут потеряться в дороге. В этом случае программа в дальнейшем всегда будет принимать неверные значения. Проблема решается добавлением таймаута, который ожидает некоторое время после получения первого байта второй байт и сбрасывает ожидание, если тот не пришёл.

Поле длины

Если длина данных неизвестна, то либо вводится поле длины, либо стартовый и стоповый байты. Для длины пакет будет выглядеть, как на рис. 2.20:

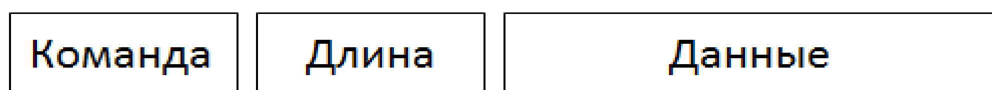

Рис. 2.20

При приёме сначала принимается команда, потом длина, а потом заданное в длине число данных. Это лучше всего делать конечным автоматом:

```
if (receive_count > 0)
{
  receive_count = 0;

  if (R_Mode == 0) {
        command = USART_ReceiveData(USARTx);
R_Mode = 1;
  } else if (R_Mode == 1) {
        Length = USART_ReceiveData(USARTx);
R_Mode = 2;
I=0;
```

```
	} else if (R_Mode < Length+2) {
		Buffer[I] = USART_ReceiveData(USARTx);
		R_Mode++;
		I++;
	} else if (R_Mode == Length+2) {
		R_Mode = 0;
		R_flag = 1;
	}
}
```

В программе по-очереди принимаются байты и сохраняются в соответствующие переменные. По окочании приёма выставляется флаг того, что данные приняты успешно. И теперь их можно обрабатывать.

Поле контроля целостности

В качестве одного из полей может быть поле контрольной суммы. Считть сумму можно как угодно. Обычно для этого используется алгоритм CRC. В данном же случае может быть достаточно просто делать XOR всех байтов пакета. Получившийся байт вставляется в самый конец пакета (Рис. 2.21).

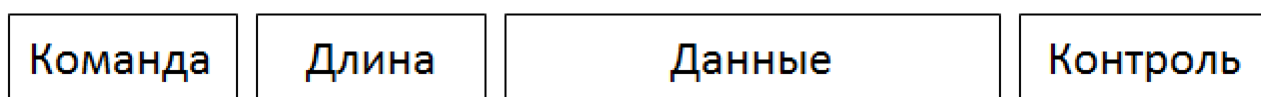

Рис. 2.21

При приёме сначала вычисляется контрольная сумма, а затем обрабатываются данные, если сумма сошлась.

Поле стоп байта

В случае, если нет возможности использовать поле длины, можно использовать стоп байт. Расположение полей в этом случае выглядит примерно так, как на рис. 2.22.

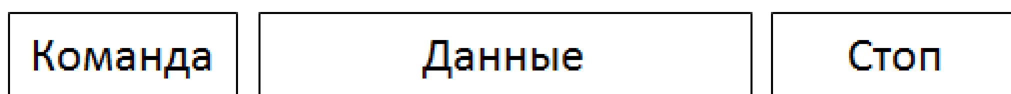

Рис. 2.22

При приёме этих пакетов, используется алгоритм, аналогичный предыдущему, только каждый раз проверяется проход конца пакета. Разумеется, символ конца пакета должен быть запрещён в поле данных. Такой вариант возможен при использовании символьного протокола, где символом конца пакета является число 0.

```
if (receive_count > 1)
{
 receive_count = 0;

 if (R_Mode == 0) {
      command = USART_ReceiveData(USARTx);
R_Mode = 1;
I=0;
 } else if (R_Mode == 1) {
      if (Buffer[I] != 0)
{
 Buffer[I] = USART_ReceiveData(USARTx);
 I++;
 } else {
      R_Mode = 2;
 }
 } else if (R_Mode == 2) {
      R_Mode = 0;
      R_flag = 1;
 }
}
```

В результате в буфере оказывается принятая строка. Если в качестве этой строки принято число, можно использовать команду atoi. Эта команда преобразовывает строку в число. Подробнее смотрите часть 3.

Поле адреса устройства

В случае, если данные одновременно передаются не на одно, а сразу на несколько устройств, необходимо ввести в пакет поле адреса. Каждому устройству присваивается свой номер и в поле адреса вписывается номер того устройства, на которое передаётся пакет. Приняв поле адреса, устройство проверяет номер. Если это не его номер, то все остальные данные игнорируются или тоже принимаются, но ничего не выполняется. Приём надо осуществить, чтобы устройство знало, когда передача пакета была завершена. Это позволит начать ожидание адреса по окончании передачи предыдущего пакета.

Если же значение поля адреса совпадает с адресом устройства, тогда происходит полноценный приём и выполнение пакета.

Само поле адреса можно использовать в качестве старт байта. Тогда пакет будет выглядеть, как на рис. 2.23.

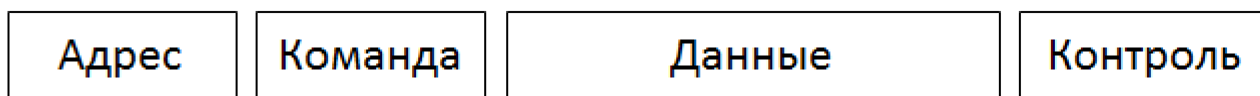

Рис. 2.23

Разработка протокола передачи данных

При разработке протокола необходимо продумать формат передачи данных так, чтобы он полностью покрывал задачи, которые необходимо решить для решения выданного задания. Причём желательно оставить возможность быстрого расширения функционала протокола. Например, ввода новых команд без переписывания большей части программы.

В начале разработки требуется определиться со структурой пакетов, передаваемых с МК на ПК и с ПК на МК. Т.к. это могут быть пакеты в которых будут содержаться разные поля.

Определившись со структурой пакетов, надо составить список команд, которые будут передаваться с ПК на МК и с МК на ПК. Для каждой команды надо описать какие данные и в каком формате будут передаваться в поле

данных с этой командой. Для каждой команды надо описать ответы на эту команду, если такое предусмотрено.

Также надо определиться будет ли МК или ПК ведущим, а другой, соответственно, ведомым или они будут передавать данные независимо.

Когда пакеты для каждого случая описаны, надо продумать алгоритм приёма этих пакетов и реакцию программы на каждую команду. Также нужен алгоритм формирования пакетов и заодно продумать способ передачи этих пакетов на ПК.

В результате при составлении описания протокола надо:

1) Описать форматы пакетов с Мк на ПК и с ПК на МК.
2) Дать описание каждой используемой команды. В это описание входит:
   a. Название команды (что надо отправить, чтобы это было распознано как команда).
   b. Функциональная нагрузка этой команды.
   c. Условия отправки этой команды (если есть).
   d. Какие данные и в каком формате входят в поле данных для этой команды.
   e. Какой ответ следует на переданную команду.
3) При инициализации системы с помощью ряда команд требуется описать последовательность отправляемых команд.
4) Описать подробно алгоритм приёма и передачи команд.

DMA

Структура DMA

Микроконтроллер stm32f4xx содержит 2 DMA. Каждое DMA разбивается на 7 потоков, которые могут работать независимо. К любому из 7-ми потоков может быть подключен любой из 7-ми каналов. Каждый канал в каждом потоке каждого DMA отвечает за свою периферию (см. рис 2.19 и 2.20). К каждому потоку может быть подключен только один канал. Копировать данные из одного буфера в другой внутри памяти МК можно любым потоком DMA.

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
Channel 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

Рис 2.19.

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
Channel 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

Рис 2.20

Программирование DMA

Перед настройкой DMA для USART, требуется настроить сам USART.

Настройка ДМА производится по стандартной схеме:

1) Включаем тактирование DMA.

2) Настраиваем DMA.

3) Если надо включаем работу DMA у периферии.

3) Когда надо включаем DMA.

4) Ждём срабатывания прерывания от DMA.

Теперь рассмотрим всё это на примере передачи данных по UART. Для этого проделаем следующие шаги:

1) Включаем тактирование DMA.

2) Включаем тактирование UART

3) Настраиваем DMA.

4) Настраиваем UART.

5) Включаем прерывания от DMA.

6) Включаем прерывания от UART.

7) Включаем общие прерывания с DMA и UART.

8) Разрешаем работу UART через DMA.

На этом инициализация заканчивается. Сам алгоритм программы будет такой:

1) Принимаем байт с ПК.

2) Включаем DMA.

3) Ждём окончания работы DMA.

4) Сбрасываем флаг окончания работы DMA.

Настроим, к примеру, передачу по DMA для USART 3.

Прежде всего, из рис. 2.1 и 2.2 узнаём какое DMA, какой поток и какой канал отвечает за передачу данных через этот USART. Обозначается в таблице это как USART3_TX. Таких комбинаций в таблице две. DMA1, поток 3, канал 4 и DMA1, поток 4, канал 7. Выберем первую комбинацию.

Включаем тактирование DMA1.

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);

Для настройки DMA существует структура DMA_InitTypeDef. Описание каждого элемента структуры представлено в табл. 3.1.

Таблица 2.6

Название	Возможные значения	Описание
DMA_Channel	DMA_Channel_0 DMA_Channel_1 DMA_Channel_2 DMA_Channel_3 DMA_Channel_4 DMA_Channel_5 DMA_Channel_6 DMA_Channel_7	Задаёт программируемый канал DMA.
DMA_PeripheralBaseAddr	Адрес регистра периферии	Задаёт 32-х битный адрес периферии, в

		который посылаются данные.
DMA_Memory0BaseAddr	Адрес начала массива	Задаёт 32-х битный адрес начала массива.
DMA_DIR	DMA_DIR_PeripheralToMemory DMA_DIR_MemoryToPeripheral DMA_DIR_MemoryToMemory	Задаёт направление передачи.
DMA_BufferSize	sizeof(buffer)	Задаёт количество передаваемых данных. В примере весь buffer.
DMA_PeripheralInc	DMA_PeripheralInc_Enable DMA_PeripheralInc_Disable	Включает или выключает увеличение на 1 значения адреса периферии на каждой итерации.
		Продолжение:
DMA_MemoryInc	DMA_MemoryInc_Enable DMA_MemoryInc_Disable	Включает или выключает увеличение на 1 значения адреса массива на каждой итерации.
DMA_PeripheralDataSize	DMA_PeripheralDataSize_Byte DMA_PeripheralDataSize_HalfWord DMA_PeripheralDataSize_Word	Задаёт размер данных в периферии.

DMA_MemoryDataSize	DMA_MemoryDataSize_Byte DMA_MemoryDataSize_HalfWord DMA_MemoryDataSize_Word	Задаёт размер данных в массиве.
DMA_Mode	DMA_Mode_Normal DMA_Mode_Circular	Задаёт режим работы: одиночный или “по кругу”.
DMA_Priority	DMA_Priority_Low DMA_Priority_Medium	Задаёт приоритет.
	DMA_Priority_High DMA_Priority_VeryHigh	
DMA_FIFOMode	DMA_FIFOMode_Disable DMA_FIFOMode_Enable	Включение-выключение FIFO.
DMA_FIFOThreshold	DMA_FIFOThreshold_1QuarterFull DMA_FIFOThreshold_HalfFull DMA_FIFOThreshold_3QuartersFull DMA_FIFOThreshold_Full	Порог заполнения FIFO.
DMA_MemoryBurst	DMA_MemoryBurst_Single DMA_ MemoryBurst _INC4 DMA_ MemoryBurst _INC8 DMA_ MemoryBurst _INC16	Определяет количество данных, передаваемых в одной посылке.
DMA_PeripheralBurst	DMA_PeripheralBurst_Single DMA_PeripheralBurst_INC4 DMA_PeripheralBurst_INC8 DMA_PeripheralBurst_INC16	Определяет количество данных, передаваемых в одной посылке.

Заполним структуру под нашу задачу. Для этого нам надо задать номер канала, по которому мы будем передавать данные, адреса, откуда копировать и куда, не забыв задать направление этого копирования. При этом адрес регистра DR USART3 должен оставаться неизменным, т.к. именно в него

должно происходить копирование. Если мы во время передачи изменим этого адрес, то все остальные данные уйдут неизвестно куда. При этом нам надо будет настроить автоматическое увеличение адреса передаваемых данных на единицу, чтобы каждый раз передавать новые данные в заданной последовательности.

```
DMA_InitTypeDef      DMAIni;
//Задаём номер канала.
DMAIni.DMA_Channel = DMA_Channel_4;
// Задаём адрес регистра USART3
DMAIni.DMA_PeripheralBaseAddr =  (uint32_t)(&USART3->DR);
//Задаём адрес первого элемента данных.
DMAIni.DMA_Memory0BaseAddr = (uint32_t)text;
//Задаём направление передачи.
DMAIni.DMA_DIR = DMA_DIR_MemoryToPeripheral;
//Задаём размер передаваемых данных. Передаём весь заполненный буфер.
DMAIni.DMA_BufferSize = sizeof(text);
/*Разумеется отключаем увеличение адреса регистра USART на единицу каждую передачу.*/
DMAIni. DMA_PeripheralInc = DMA_PeripheralInc_Disable;
/*Включаем автоматическое увеличение адреса передаваемых данных на единицу каждую передачу */
DMAIni. DMA_MemoryInc =  DMA_MemoryInc_Enable;
//Передаём по одному байту.
DMAIni. DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
//Передаём по одному байту.
DMAIni. DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
//Передаём однократно.
DMAIni. DMA_Mode = DMA_Mode_Normal;
// приоритет - средний
```

```
DMAIni. DMA_Priority = DMA_Priority_Medium;
// FIFO не используем.
DMAIni. DMA_FIFOMode = DMA_FIFOMode_Disable;
// Задаём порог FIFO – целиком.
DMAIni. DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
// Передаём по одному слову за посылку.
DMAIni. DMA_MemoryBurst = DMA_MemoryBurst_Single;
// Передаём по одному слову за посылку.
DMAIni. DMA_PeripheralBurst = DMA_MemoryBurst_Single;
```

Передаём заполненную структуру в функцию:

```
DMA_Init(DMA1_Stream3, &DMA_Init_struct);
```

Включаем разрешение использования DMA в USART. Эта функция расположена в файлах, отвечающих за USART.

```
USART_DMACmd(USART3, USART_DMAReq_Tx, ENABLE);
```

Включаем общие прерывания от DMA1. Название прерывания можно найти в файле stm32f4xx.h

```
NVIC_EnableIRQ(DMA1_Stream3_IRQn);
```

Включаем прерывания по окончанию передачи.

```
DMA_ITConfig(DMA1_Stream3, DMA_IT_TC, ENABLE);
```

Когда надо начать передачу, используем функцию:

```
DMA_Cmd(DMA1_Stream3, ENABLE);
```

По окончанию передачи сработает прерывание:

```
void DMA1_Stream3_IRQHandler(void)
{
   DMA_ClearFlag(DMA1_Stream3, DMA_FLAG_TCIF3);
}
```

В этом прерывании требуется очистить флаг прерывания, иначе оно будет выскакивать бесконечно и повесит приложение.

Перед включением DMA следует проверить его занятость. Это можно сделать, устанавливая в 1 глобальную переменную при начале передачи и сбрасывая её в прерывании. Например:

```
uint8_t DMA_Work_En=0;

void DMA1_Stream3_IRQHandler(void)
{
        DMA_ClearFlag(DMA1_Stream3, DMA_FLAG_TCIF3);
  DMA_Work_En = 0;
}



int main(void)
{
        while(DMA_Work_En) {}/*перед новой отправкой ждём окончания старой*/
  DMA_Cmd(DMA1_Stream3, ENABLE);//Включаем DMA
        DMA_Work_En = 1;//Устанавливаем флаг.
}
```

Если надо сменить размер передаваемых данных, то надо сначала выключить DMA, перенастроить его, переписав строку:

DMAIni.DMA_BufferSize = sizeof(text);

И повторно вызвав функцию:

DMA_Init(DMA1_Stream3, &DMA_Init_struct);

Пример настройки DMA

Расмотрим настройку DMA для USART1. Предположим, что по таблицам его вывод TX расположен на канале 1 первого потока DMA1.

```
DMA_InitTypeDef    DMA_Init_struct;
```

```
uint8_t text[256]={0};

// Включаем тактирование
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);

// Указываем канал 1
DMA_Init_struct.DMA_Channel = DMA_Channel_1;
//Записываем указатель куда…
DMA_Init_struct.DMA_PeripheralBaseAddr = (uint32_t)(&USART1->DR);
//Записываем указатель откуда
DMA_Init_struct.DMA_Memory0BaseAddr = (uint32_t)text;
//Пишем от какой области памяти к какой
DMA_Init_struct.DMA_DIR = DMA_DIR_MemoryToPeripheral;
//Пишем сколько передавать (Тут весь буфер)
DMA_Init_struct.DMA_BufferSize = sizeof(text);
//Говорим не увеличивать указатель периферийного адреса на единицу
DMA_Init_struct.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
//Говорим увеличивать указатель адреса памяти на единицу
DMA_Init_struct.DMA_MemoryInc = DMA_MemoryInc_Enable;
//Говорим, что данные в периферии однобайтные
DMA_Init_struct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
//Говорим, что данные в памяти однобайтные
DMA_Init_struct.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
//Выбираем одиночную передачу
DMA_Init_struct.DMA_Mode = DMA_Mode_Normal;
//Указываем приоритет
DMA_Init_struct.DMA_Priority = DMA_Priority_Medium;
//Выключаем ФИФО
```

```
DMA_Init_struct.DMA_FIFOMode = DMA_FIFOMode_Disable;
DMA_Init_struct.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
//Потоковая передача по одному
DMA_Init_struct.DMA_MemoryBurst = DMA_MemoryBurst_Single;
DMA_Init_struct.DMA_PeripheralBurst = DMA_MemoryBurst_Single;
// Инициализируем DMA
DMA_Init(DMA1_Stream1, &DMA_Init_struct);
// Указываем работу DMA с передачей USART1
USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);
// Включаем прерывания по окончанию работы DMA
NVIC_EnableIRQ(DMA1_Stream1_IRQn);
DMA_ITConfig(DMA1_Stream1, DMA_IT_TC, ENABLE);
```

Теперь можно передать все данные из буфера text по USART одной строкой:

```
DMA_Cmd(DMA1_Stream1, ENABLE);
```

И обрабатывать итог в прерывании:

```
//-----------------------------------------------------
// DMA interrupt
//-----------------------------------------------------
void DMA1_Stream1_IRQHandler(void)
{
   DMA_ClearFlag(DMA1_Stream1, DMA_FLAG_TCIF1);
}
```

SPI

Общая настройка SPI

Для настройки SPI прежде всего требуется определиться с числом выводов, которые будут использоваться для работы. Минимально нужен 1 вывод для передачи данных, если вы собираетесь генерировать ШИМ сигнал. Но обычно используется 4 вывода: Это выбор микросхемы, тактовый вывод и 2 вывода передачи данных.

Все эти выводы настраиваются на альтернативную функцию за исключением выбора микросхемы. Этот вывод можно либо настроить на аппаратную реализацию или устанавливать программно, т.к. аппаратная часть не всегда работает так как требуется реализуемому протоколу общения с внешней микросхемой. Если вывод выбора микросхемы управляется программно, но его надо настроить как выход.

Не надо забывать включать тактирование для всех используемых портов и самого SPI. И не забывайте подключать альтернативную функцию для выводов в режиме альтернативной функции:

GPIO_PinAFConfig(GPIOx, GPIO_PinSourceY, GPIO_AF_SPIz);

Где x – имя порта. Y – имя вывода и z – номер SPI.

Остальные настройки порта были рассмотрены ранее.

Для настройки SPI требуется настроить структуру с типом:

SPI_InitTypeDef

Наиболее важными полями в ней являются SPI_CPOL и SPI_CPHA, определяющие уровень сигнала, когда тактовая не передаётся и номер среза с которого снимаются данные (см. рис. 2.21).

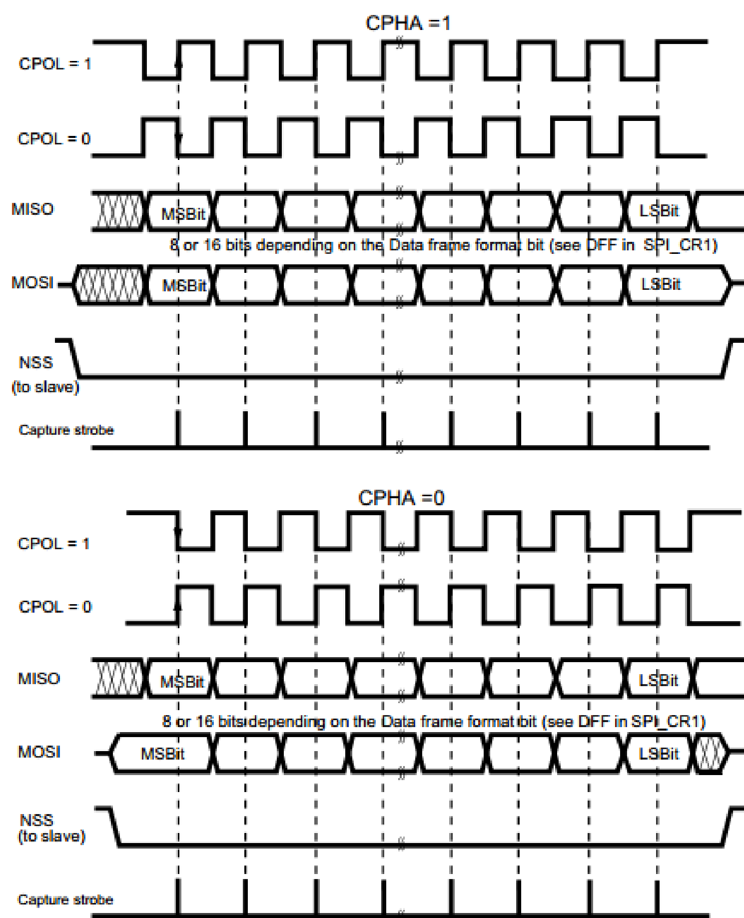

Рис. 2.21

Из рис. 2.21 можно видеть когда снимаются даннные при разных установках уровня и фазы. Вертикальные полосы показывают момент снятия данных со входа. При настройке надо установить эти параметры так, что бы временной график сигнала, показанного на рис. 2.21 совпал с тем, что представлен в документации.

Далее надо настроить делитель частоты и направление выхода данных (начиная с младшего или старшего).

Теперь рассмотрим настройку SPI подробнее на примере работы с микросхемой акселерометра.

Пример подключения микросхемы по SPI

Приведём пример настройки SPI для общения с микросхемой акселерометра LIS302DL. Прежде всего, на схеме необходимо найти к каким выводам подключен датчик (рис 2.23).

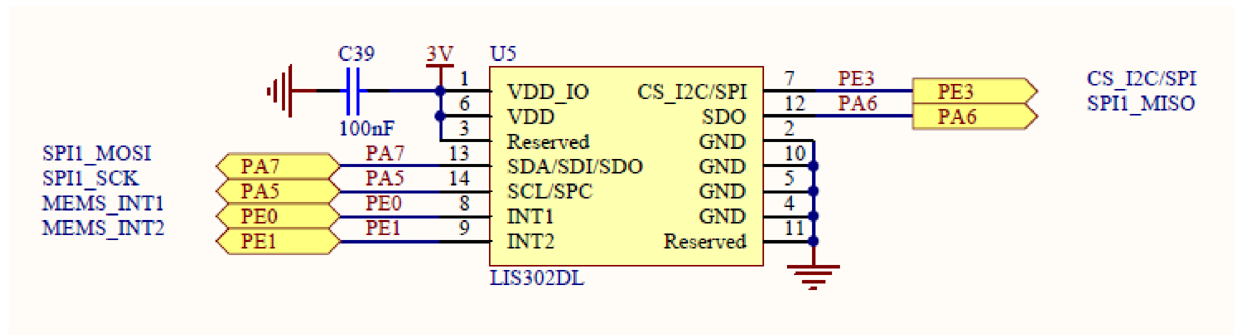

Рис 2.23

Как можно видеть, выводы SPI подключены к выводам 5-7 порта А. При этом Chip select (CS) подключен к выводу E3. На эти ножки микроконтроллера выведен SPI1. Пример настройки этих выводов представлен ниже.

```
GPIO_InitTypeDef GPIO_Init_SPI;

//Включаем тактирование выводов
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

//Подключаем альтернативную функцию.
GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);

GPIO_Init_SPI.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
GPIO_Init_SPI.GPIO_Mode = GPIO_Mode_AF;
GPIO_Init_SPI.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init_SPI.GPIO_OType = GPIO_OType_PP;
GPIO_Init_SPI.GPIO_PuPd = GPIO_PuPd_DOWN;

GPIO_Init(GPIOA, &GPIO_Init_SPI);
```

```
GPIO_Init_SPI.GPIO_Pin = GPIO_Pin_3;
GPIO_Init_SPI.GPIO_Mode = GPIO_Mode_OUT;
GPIO_Init_SPI.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init_SPI.GPIO_OType = GPIO_OType_PP;
GPIO_Init_SPI.GPIO_PuPd = GPIO_PuPd_NOPULL;

GPIO_Init(GPIOE, &GPIO_Init_SPI);

GPIO_SetBits(GPIOE,GPIO_Pin_3);
```

После настройки выводов, необходимо настроить сам SPI. Но перед этим следует определиться с рядом параметров, которые будут устанавливаться при настройке SPI. Самые главные из них CPOL и CPHA. Для определения этих параметров надо найти временные диаграммы сигналов SPI, воспринимаемых датчиком, т.к. существует 4 режима работы SPI. Рассмотрим тактовый сигнал SPC для датчика LIS302DL (рис 2.21).

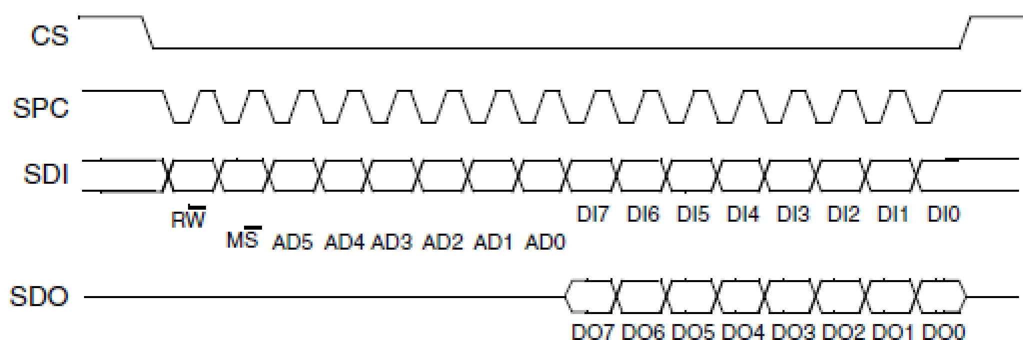

Рис 2.21

Отметим, что в не рабочем режиме этот сигнал принимает значение 1, а считывание данных при передаче происходит по второму фронту.

Также важна скорость передачи данных, т.к. зачастую микросхемы не воспринимают скорости выше и ниже которых границ. При этом SPI тактируется от некоторой шины и скорость передачи не задаётся в явном

виде. Она определяется делителем. Важно, что бы частота тактового сигнала на выходе делителя находилась в заданном диапазоне.

Также стоит определиться с числом передаваемых бит данных за одну посылку. Обычно это число может быть равно 8 или 16, но бывают варианты, когда требуется передавать отличное от этих чисел число бит данных. Важно, что бы ведущее устройство умело это делать. В нашем случае требуется передавать за одну посылку 1 байт данных.

Также важным параметром является порядок следования бит данных. Они могут идти от старшего к младшему или от младшего к старшему. Это тоже надо выяснять в технической документации на микросхему.

Вывод выбора микросхемы можно установить аппаратным или программным.

```
SPI_InitTypeDef SPI_InitStructure;

RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_Init(SPI1, &SPI_InitStructure);

SPI_Cmd(SPI1, ENABLE);

SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Set);
```

Рассмотрим настройки подробнее:

Direction – Определяет направление и количество выводов.

DataSize – Размер передаваемого слова.

CPOL – Уровень сигнала, который устанавливается пока не ведётся передача.

CPHA – Номер среза по которому идёт чтение данных.

NSS – выбирает между программным и аппаратным выбором датчика (CS)

BaudRatePrescaler – определяет предделитель частоты.

FirstBit – Определяет порядок следования битов: от старшего к младшему или наоборот.

Mode – режим работы: ведущий или ведомый.

Передача и приём данных происходит одновременно. Но в любом случае надо сначала послать запрос и только затем по второму запросу можно получить данные. При этом сигнал CS должен оставаться в 0.

Функция одной передачи выглядит следующим образом:

```
uint8_t writeData(uint8_t data)
{
	while(SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == SET)
{};
	SPI_I2S_SendData(SPI1, data);
	while(SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == SET)
{};
	return SPI_I2S_ReceiveData(SPI1);
}
```

Описанные выше действия позволяют передать на микросхему и получить с микросхемы 1 байт данных.

На этом уровень SPI заканчивается. Всё описанное выше аналогичным образом настраивается при подключении любой микросхемы. Требуется только правильно установить параметры.

Реализация протокола микросхемы акселерометра

Дальнейшие действия определяются алгоритмом общения, который реализован в самой микросхеме и уже не зависит непосредственно от SPI.

В микросхеме LIS302DL есть ряд регистров, отвечающих за тот или иной параметр. Каждый регистр имеет свой адрес. Для чтения из регистра надо сначала передать адрес этого регистра с установленным старшим битом в 1, что значит, что далее будет чтение из этого регистра. А потом передать нулевые данные для получения содержимого запрашиваемого регистра.

Для записи же сначала передаётся адрес, а потом данные, которые в регистр надо записать.

Чтение из регистра будет выглядеть следующим образом:

```
uint8_t getReg(uint8_t address)
 {
        uint8_t data=0;
        address|=(1<<7);
        GPIO_ResetBits(GPIOE,GPIO_Pin_3);
        writeData(address);
        data = writeData(0x00);
        GPIO_SetBits(GPIOE,GPIO_Pin_3);
        return data;
}
```

Отметим, что бит, говорящий о типе операции чтение-запись – это старший бит передаваемого слова. Потому в функции чтения регистра есть строка:

```
address|=(1<<7);
```

Она устанавливает старший бит в единицу, говорящую, что будет производиться чтение данных.

Запись в регистр выглядит так:

```
void setReg(uint8_t address, uint8_t value)
{
	GPIO_ResetBits(GPIOE,GPIO_Pin_3);
	writeData(address);
	writeData(value);
	GPIO_SetBits(GPIOE,GPIO_Pin_3);
}
```

Ну и зажигание светодиода в зависимости от направления наклона показано ниже:

```
int main(void)
{
	LEDs_ini();
	SPI_ini();
	SysTick_Config(SystemCoreClock/1000);//1 ms
	setReg(0x20, 0x47);
	while(1)
	{
		SPI_data = getReg(LIS302DL_OUT_X);
		if (SPI_data > 10) {
	BLUE_ON();YELLOW_OFF();
} else if (SPI_data < -10)   {
```

```
        BLUE_OFF(); YELLOW_ON();
} else {
        YELLOW_OFF();BLUE_OFF();
}
                SPI_data = getReg(LIS302DL_OUT_Y);
                if (SPI_data > 10) {
        RED_ON();GREEN_OFF();
} else if (SPI_data < -10)   {
        RED_OFF(); GREEN_ON();
        } else {
        RED_OFF();GREEN_OFF();
        }
                SPI_data = getReg(LIS302DL_OUT_Z);
                delay_ms(100);
        }
}
```

Вместо LIS302DL_OUT_x надо подставить адреса регистров, которые хранят данные ускорений.

ЖКИ

Подключение дисплея к контроллеру

Интерфейс подключения – параллельный. Для соединения индикатора с микроконтроллером используется 11 линий — восемь для передачи данных (D0 - D7) и три линии управления (RS, E, R/W). Линия RS служит для сообщения контроллеру индикатора о том, что именно передается по шине: команда или данные (RS = 1 — данные, RS = 0 — команда). По линии E передается строб-сигнал, сопровождающий запись или чтение данных: по переходу сигнала на линии E из 1 в 0 осуществляется запись данных во входной буфер микроконтроллера индикатора. Запись информации в ЖКИ происходит по спаду этого сигнала. Потенциал на управляющем выводе R/W (Read/Write) задает направление передачи информации, при R/W = 0 осуществляется запись в память индикатора, при R/W = 1 – чтение из нее. Еще три линии предназначены для подачи питающего напряжения (VDD, GND) и напряжения смещения, которое управляет контрастностью дисплея.

Восьми битное подключение.

Стандартное подключение дисплея приведено на рис 2.21. Этот способ подключения использует всю восьми битную шину данных.

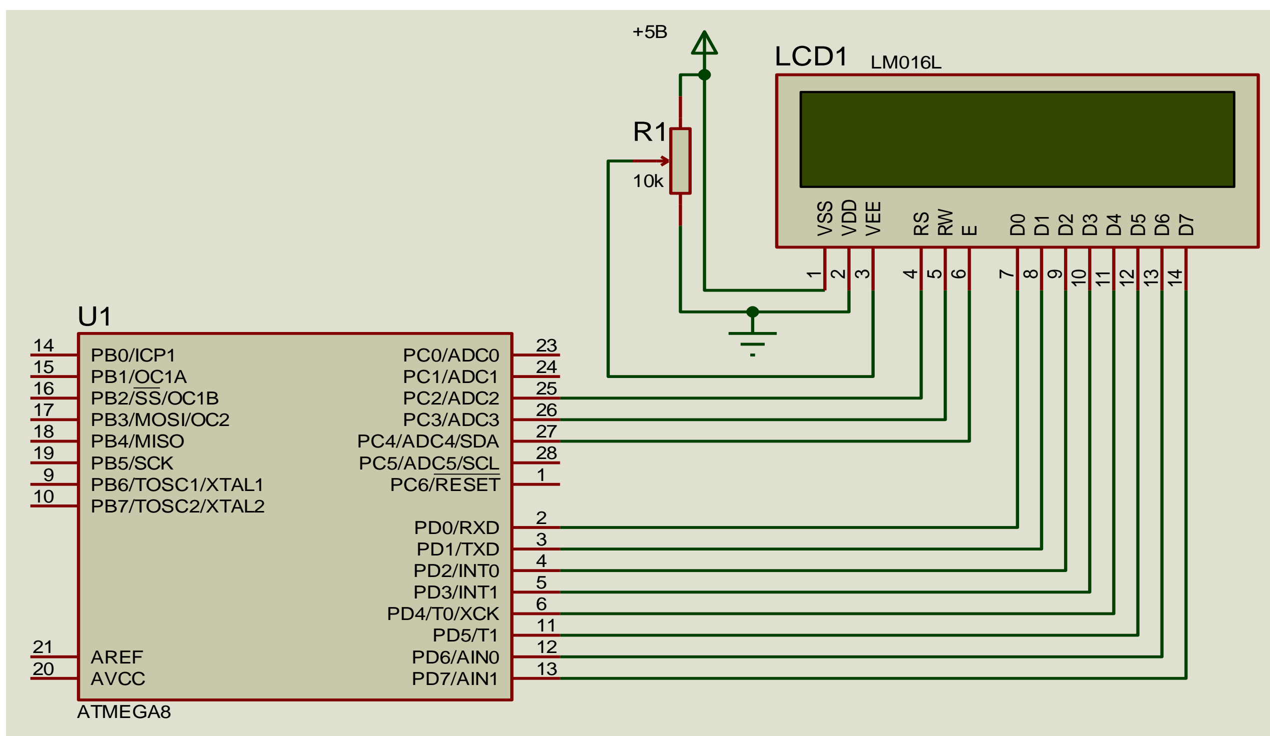

Рис.2.21 – Схема подключения ЖКИ к микроконтроллеру (8–ми битный интерфейс передачи данных)

При использовании 8-ми битного интерфейса передачи данных у контроллера используется 11 выводов, для уменьшения используемых выводов контроллера, существует 4-х битный интерфейс передачи данных.

4 битная шина

В данном случае для соединения индикатора с микроконтроллером используется 7 линий — четыре для передачи данных (D4 - D7) и три линии управления (RS, E, R/W). Данное подключение приведено на рис 2.22.

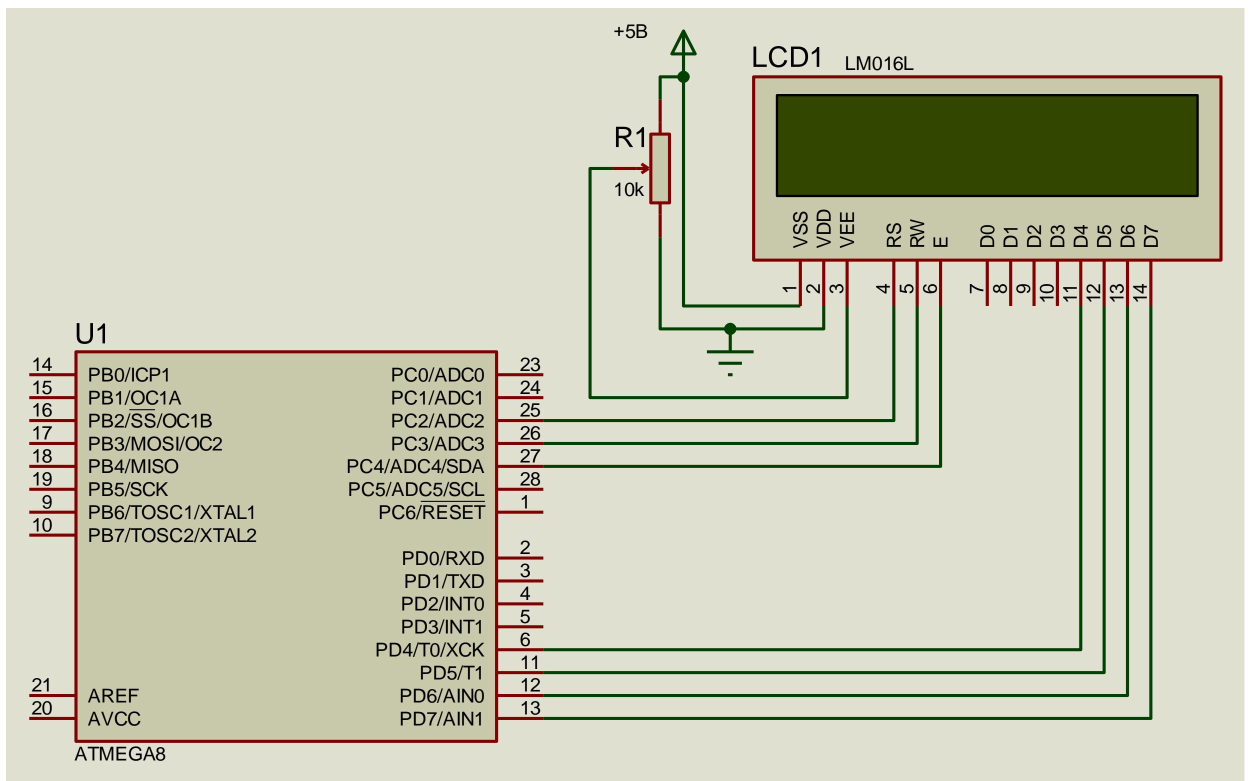

Рис.2.22 Схема включения дисплея

При таком типе подключения ЖКИ, уменьшается количество используемых выводов у контроллера, однако при этом замедляется передача информации, так как данные отправляются в два этапа.

Контроллер HD44780

Для вывода информации на экран используется контроллер дисплея HD44780, преобразующий команды с микроконтроллера в понятные дисплею электронные сигналы. Для вывода информации на дисплей требуется отправлять соответствующие команды контроллеру. Список команд, понимаемых контроллеров представлен в табл 2.1.

Табл. 2.1

<table>
<tr><th rowspan="3">Название инструкции</th><th colspan="10">Состояние выводов</th><th rowspan="3">Время</th></tr>
<tr><th rowspan="2">RS</th><th rowspan="2">R/W</th><th colspan="4">Старшие биты</th><th colspan="4">Младшие биты</th></tr>
<tr><th>DB7</th><th>DB6</th><th>DB5</th><th>DB4</th><th>DB3</th><th>DB2</th><th>DB1</th><th>DB0</th></tr>
<tr><td>Очистка дисплея</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1.52мс</td></tr>
<tr><td>Возврат в начальную позицию</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>*</td><td>1.52мс</td></tr>
<tr><td>Режим ввода</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>I/D</td><td>S</td><td>37мкс</td></tr>
<tr><td colspan="12">I/D = 1: адрес DDRAM увеличивается, I/D =0: уменьшается
S = 1: сдвиг рабочей области дисплея по DDRAM разрешен, S=0: нет</td></tr>
<tr><td>Вкл/выкл дисплея и курсора</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>D</td><td>C</td><td>B</td><td>37мкс</td></tr>
<tr><td colspan="12">D = 1: дисплей (изображение) включен, D=0: выключен.
C = 1: курсор включен, C = 0: выключен.
B = 1: мерцание курсора включено, B = 0: выключено</td></tr>
<tr><td>Сдвиг курсора или видимой области дисплея</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>S/C</td><td>R/L</td><td>*</td><td>*</td><td>37мкс</td></tr>
<tr><td colspan="12">S/C = 1: сдвинуть дисплей, S/C = 0: переместить курсор
R/L = 1: вправо, R/L = 0: влево</td></tr>
<tr><td>Начальные установки</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>DL</td><td>N</td><td>F</td><td>*</td><td>*</td><td>43мкс</td></tr>
<tr><td colspan="12">DL=1 установка интерфейса 8-и битный, DL=0: 4-х битный</td></tr>
</table>

Табл. 2.1

Название инструкции	Состояние выводов										Время
	RS	R/W	Старшие биты				Младшие биты				
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
N=1: 2 строки, N= 0: 1 строка F=1: матрица символа 5x10, F= 0: 5x8											
Установка адреса CGRAM	0	0	0	1	ACG5	ACG4	ACG3	ACG2	ACG1	ACG0	43мкс

Табл. 2.1 (продолжение)

Установка адреса DDRAM	0	0	1	ADD6	ADD5	ADD4	ADD3	ADD2	ADD1	ADD0	37мкс
Чтение флага Занятости (BF) и адреса	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	0мкс
Запись данных в ОЗУ	1	0	D7	D6	D5	D4	D3	D2	D1	D0	37мкс
Чтение данных из ОЗУ	1	1	D7	D6	D5	D4	D3	D2	D1	D0	37мкс

В таблице показано какие значения надо установить на управляющих выводах и какие значения подать на шину.

Прежде чем вы сможете использовать данный контроллер, вам придётся его сначала инициализировать.

Команда Установка адреса DDRAM устанавливает курсор в нужное место дисплея. Адреса ячеек в зависимости от строки можно посмотреть в табл. 2.2.

Таблица 2.2

Позиция дисплея	1	2	3	4	………………………	39	40

Адрес DDRAM	0x00	0x01	0x02	0x03	……………………….	0x26	0x27
	0x40	0x41	0x42	0x43	……………………….	0x66	0x67

Инициализация контроллера.

В зависимости от типа подключения есть 2 различных способа инициализации.

8-ми битный интерфейс передачи данных

Включение питания дисплея

Задержка (Более 15мс пока Vcc установится до 4,5В)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	BF нельзя прочесть до этой
0	0	0	0	1	1	*	*	*	*	операции (интерфейс 8 бит)

Задержка (более 4,5мс)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	BF нельзя прочесть до этой
0	0	0	0	1	1	*	*	*	*	операции (интерфейс 8 бит)

Задержка (более 100мкс)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	BF нельзя прочесть до этой
0	0	0	0	1	1	*	*	*	*	операции (интерфейс 8 бит)

Теперь можно читать BF или выставлять задержку между посылками данных о ЖКИ более, чем время исполнения команд контроллера ЖКИ (Таблица 5)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Установка режима дисплея
0	0	0	0	1	DL	N	F	*	*	(Function set)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Вкл/выкл дисплея и курсора
0	0	0	0	0	0	1	D	C	B	(Display on/off control)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Очистка дисплея
0	0	0	0	0	0	0	0	0	1	(Clear display)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Смещение курсора дисплея
0	0	0	0	0	0	0	1	I/D	S	(Entry mode set)

4-х битная шина

Включение питания дисплея

Задержка (Более 15мс пока Vcc установится до 4,5В)

RS	RW	DB7	DB6	DB5	DB4	BF нельзя прочесть до этой операции
0	0	0	0	1	1	Function set (интерфейс 8 бит)

Задержка (более 4,5мс)

RS	RW	DB7	DB6	DB5	DB4	BF нельзя прочесть до этой операции
0	0	0	0	1	1	Function set (интерфейс 8 бит)

Задержка (более 100мкс)

RS	RW	DB7	DB6	DB5	DB4	BF нельзя прочесть до этой операции
0	0	0	0	1	1	Function set (интерфейс 8 бит)

Теперь можно читать BF или выставлять задержку между посылками данных о ЖКИ более, чем время исполнения команд контроллера ЖКИ (Таблица 5)

RS	RW	DB7	DB6	DB5	DB4	Установка режима дисплея (Function set)
0	0	0	0	1	DL	Переход в 4 битный режим (DL=0)

Далее отправляем команду в два этапа. Сначала старшие биты, а затем младшие (Таблица 5)

RS	RW	DB7	DB6	DB5	DB4	Установка режима дисплея (Function set)
0	0	0	0	1	DL	
RS	RW	DB7	DB6	DB5	DB4	
0	0	N	F	0	0	

RS	RW	DB7	DB6	DB5	DB4	Вкл/выкл дисплея и курсора

0	0	0	0	0	0	(Display on/off control)
RS	RW	DB7	DB6	DB5	DB4	
0	0	1	D	C	B	

RS	RW	DB7	DB6	DB5	DB4	
0	0	0	0	0	0	Очистка дисплея
RS	RW	DB7	DB6	DB5	DB4	(Clear display)
0	0	0	0	0	1	

RS	RW	DB7	DB6	DB5	DB4	
0	0	0	0	0	0	Смещение курсора дисплея
RS	RW	DB7	DB6	DB5	DB4	(Entry mode set)
0	0	0	1	I/D	S	

Программирование дисплея

Переопределение выводов

Выводов, подключенных к микроконтроллеру – одиннадцать. Поэтому и переопределить требуется одиннадцать выводов. Это позволит не вылавливать по всей программе номера выводов, при подключении дисплея к другим выводам. Сделать это можно, например, следующим образом:

```
#define RW(a)     GPIO_WriteBit(GPIOB, GPIO_Pin_4, (BitAction)a)
#define RS(a)     GPIO_WriteBit(GPIOB, GPIO_Pin_3, (BitAction)a)
#define EN(a)     GPIO_WriteBit(GPIOB, GPIO_Pin_5, (BitAction)a)
#define DB7(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_14, (BitAction)a)
#define DB6(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_13, (BitAction)a)
#define DB5(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_12, (BitAction)a)
#define DB4(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_11, (BitAction)a)
#define DB3(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_8, (BitAction)a)
#define DB2(a)    GPIO_WriteBit(GPIOB, GPIO_Pin_9, (BitAction)a)
```

```
#define DB1(a)       GPIO_WriteBit(GPIOB, GPIO_Pin_7, (BitAction)a)
#define DB0(a)       GPIO_WriteBit(GPIOB, GPIO_Pin_6, (BitAction)a)
```

В данном случае использовалась функция GPIO_WriteBit, позволяющая устанавливать самостоятельно значение на выводе. Чтобы установить единицу, к примеру, на выводе EN требуется написать:

```
EN(1);
```

Чтобы записать байт на шину, можно воспользоваться функцией:

```
void write_data(uint8_t data)
{
        if (((data>>7)&0x01) == 0x01){DB7(1);} else {DB7(0);}
        if (((data>>6)&0x01) == 0x01){DB6(1);} else {DB6(0);}
        if (((data>>5)&0x01) == 0x01){DB5(1);} else {DB5(0);}
        if (((data>>4)&0x01) == 0x01){DB4(1);} else {DB4(0);}

        if (((data>>3)&0x01) == 0x01){DB3(1);} else {DB3(0);}
        if (((data>>2)&0x01) == 0x01){DB2(1);} else {DB2(0);}
        if (((data>>1)&0x01) == 0x01){DB1(1);} else {DB1(0);}
        if (((data>>0)&0x01) == 0x01){DB0(1);} else {DB0(0);}
}
```

Если же используется 4-х битная система подключения, то функция подключения упрощается:

```
void write_data(uint8_t data)
{
        if (((data>>3)&0x01) == 0x01){DB7(1);} else {DB7(0);}
        if (((data>>2)&0x01) == 0x01){DB6(1);} else {DB6(0);}
        if (((data>>1)&0x01) == 0x01){DB5(1);} else {DB5(0);}
        if (((data>>0)&0x01) == 0x01){DB4(1);} else {DB4(0);}
 }
```

В данном случае в 4 старших бита шины данных записываются только 4 младших бита от переданного в функцию байта.

Функции передачи команды и данных.

Для передачи команды на дисплей требуется выполнить следующие действия:

1) Установить RS в 0.
2) Установить на шине нужное значение.
3) Установить EN в 1.
4) Подождать некоторое время. Например, подождать пока бит BF не установится в 0.
5) Установить EN в 0.

Для 4-х битной щины надо передать сначала старший полубайт, затем младший.

Для передачи данных используется точно такая же функция, за исключением того, что сигнал RS устанавливается равным 1.

Запрограммировать эти функции можно, например, так:

```
void LCD_SendCommand(uint8_t Command)
{
        uint16_t i;

        RS(0);
        write_data(Command);
        EN(1);
        for (i=0;i<1000;i++){}
        EN(0);
}

void LCD_SendData(uint8_t Command)
{
        uint16_t i;

        RS(1);
```

```
        write_data(Command);
        EN(1);
        for (i=0;i<1000;i++){}
        EN(0);
}
```

Инициализация контроллера.

Следуя описанной выше схеме, можно написать следующую программу:

```
delay_ms(17);
RW(0);
LCD_SendCommand(0x30);
delay_ms(5);
LCD_SendCommand(0x30);
delay_ms(1);
LCD_SendCommand(0x30);
delay_ms(1);
LCD_SendCommand(0x38);
delay_ms(1);
LCD_SendCommand(0x0F);
delay_ms(1);
LCD_SendCommand(0x01);
delay_ms(1);
LCD_SendCommand(0x06);
delay_ms(1);
LCD_SendCommand(0x02);
delay_ms(1);
```

Разумеется, эта инициализация верна для 8-мибитной шины.

Запись строки данных

Запись строки стандартна. Передаём данные, пока не достигнем 0. Выглядит это так:

```
void LCD_SendString(char *strTemp)
{
	uint8_t i=0;

	while(strTemp[i] != 0)
	{
		LCD_SendData(strTemp[i]);
		i++;
	}
}
```

АЦП

Настройка регулярного канала.

Рассмотрим настройку регулярного канала АЦП. Настроим АЦП на ножке A4. Прежде всего, надо узнать какие АЦП имеют доступ к этой ножке и какие каналы на неё выведены. В частности это 4-й канал первого АЦП.

Как обычно используем стандартную схему:

1) Включить тактирование порта
2) Настроить вывод
3) Включить тактирование АЦП
4) Настроить АЦП
5) Включить нужные прерывания
6) Включить глобальные прерывания
7) Включить АЦП

При настройке порта главное в режиме задать аналоговый режим.

```
GPIO_InitTypeDef     GPIO_Init_user;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_Init_user.GPIO_Pin = GPIO_Pin_4;
GPIO_Init_user.GPIO_Mode = GPIO_Mode_AN;
GPIO_Init_user.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init_user.GPIO_OType = GPIO_OType_PP;
GPIO_Init_user.GPIO_PuPd = GPIO_PuPd_NOPULL;

GPIO_Init(GPIOA, & GPIO_Init_user);
```

Включаем тактирование АЦП:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

Настраиваем:

```
ADC_InitTypeDef      ADC_InitType;

ADC_InitType.ADC_ContinuousConvMode = DISABLE;
ADC_InitType.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitType.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
ADC_InitType.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
ADC_InitType.ADC_NbrOfConversion = 1;
ADC_InitType.ADC_Resolution = ADC_Resolution_12b;
ADC_InitType.ADC_ScanConvMode = DISABLE;

ADC_Init(ADC1, &ADC_InitType);
```

Рассмотрим настройки подробнее:

ContinuousConvMode – Этот режим, если включен, запускает следующее преобразование сразу по окончании предыдущего. Так можно добиться максимальной скорости работы АЦП. В нашем случае это не надо и данная функция отключена.

DataAlign – выравнивание данных в 2-хбайтном слове. Есть 2 варианта. ADC_DataAlign_Right при котором данные выравниваются по правому краю, а неиспользуемые биты при этом равны нулю. Т.е. мы получаем обычные числа в 2-х байтах от 0 до 8192. При ADC_DataAlign_Left данные выравниваются по левому краю. Т.е. фактически для 12-ти битного преобразования они увеличиваются в 16 раз. Это может быть использовано например при передаче их через SPI, поддерживающий 12-ти битную передачу данных. Если настроить SPI на передачу начиная со старшего разряда.

ExternalTrigConvEdge – настраивает запуск преобразования по какому либо событию, например переполнению таймера. В нашем случае не требуется.

ExternalTrigConv – Устанавливает какие именно события запустят АЦП. Т.к. триггер отключен, то эта функция не используется.

NbrOfConversion – число каналов, которые будет сканировать МК. Сюда записывается требуемое значение, а ниже, если это число больше 1 и ADC_ ScanConvMode=ENABLE, описывается какие каналы и в какой последовательности они будут сканироваться

ScanConvMode – Этот параметр определяет будет ли АЦП сканировать несколько каналов. Если этот режим включен, то АЦП будет последовательно оцифровывать данные с заданных каналов в заданной последовательности. И каналы и последовательность легко можно задать. Но возникает небольшая проблема со снятием данных.

Настраиваем конкретный канал. В нашем случае это всего один канал, потому настройка будет выглядеть так:

```
ADC_RegularChannelConfig(ADC1,ADC_Channel_4,1,
DC_SampleTime_56Cycles);
```

Из параметров тут:

ADC1 – номер настраиваемого АЦП.

ADC_Channel_4 задаёт снимаемый канал.

1 – так называемый rank. Показывает в каком порядке этот канал будет оцифровываться. В нашем случае канал один, потому и rank=1.

DC_SampleTime_56Cycles – задаёт за какое время будет произведена оцифровка. Чем медленнее, тем точнее.

Теперь осталось настроить прерывания и включить:

```
NVIC_EnableIRQ(ADC_IRQn);
ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

ADC_Cmd(ADC1, ENABLE);
```

На этом настройка закончена.

Чтобы запустить преобразование, используйте функцию:

```
ADC_SoftwareStartConv(ADC1);
```

По окончании преобразования программа попадёт в функцию прерывания:

```
void ADC_IRQHandler(void)
{
	ADC_ClearFlag(ADC1, ADC_FLAG_EOC);
	ADC_result = ADC_GetConversionValue(ADC1);
}
```

Сбрасываем флаг и считываем результат преобразования.

Пример настройки АЦП

Настроим АЦП1. Пусть оно расположено на выводе A0 и является каналом 1.

```
GPIO_InitTypeDef     GPIO_Init_user;
ADC_InitTypeDef        ADC_InitType;

// Включаем тактирование порта
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

//Настраиваем порт как аналоговый
GPIO_Init_user.GPIO_Pin = GPIO_Pin_0;
GPIO_Init_user.GPIO_Mode = GPIO_Mode_AN;
GPIO_Init_user.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init_user.GPIO_OType = GPIO_OType_PP;
GPIO_Init_user.GPIO_PuPd = GPIO_PuPd_NOPULL;

GPIO_Init(GPIOA, & GPIO_Init_user);

// Включаем тактирование АЦП
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

//Выключаем повторный запуск по окончании преобразования
ADC_InitType.ADC_ContinuousConvMode = DISABLE;
//Выравнивание по правому краю
ADC_InitType.ADC_DataAlign = ADC_DataAlign_Right;
//Триггер выключен
ADC_InitType.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
```

```
ADC_InitType.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
//Преобразование при сканировании одно
	ADC_InitType.ADC_NbrOfConversion = 1;
	//Разрешение преобразования в 12 бит
	ADC_InitType.ADC_Resolution = ADC_Resolution_12b;
	//Сканирование выключено
	ADC_InitType.ADC_ScanConvMode = DISABLE;

	//Записываем в АЦП настройки
	ADC_Init(ADC1, &ADC_InitType);



	//Пишем с какого канала читать
	ADC_RegularChannelConfig(ADC1,ADC_Channel_1,1,
DC_SampleTime_56Cycles);

	//Включаем прерывания
	NVIC_EnableIRQ(ADC_IRQn);
	ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

	//Включаем АЦП
	ADC_Cmd(ADC1, ENABLE);
```

Теперь можно считывать данные с помощью строки:

```
	//Считываем данные
	ADC_SoftwareStartConv(ADC1);
```

По окончании преобразования программа попадёт в функцию прерывания:

```
//Функция прерывания
void ADC_IRQHandler(void)
{
	ADC_ClearFlag(ADC1, ADC_FLAG_EOC);
	ADC_result = ADC_GetConversionValue(ADC1);
}
```

Программирование ЦАП

Как обычно, сначала включается тактирование ЦАП порта, затем заполняются структуры ЦАП и порта. Эти структуры используются при вызове инициализирующих функций. В конце включается ЦАП. На этом настройка на самом простом уровне заканчивается.

Начнём. Прежде всего, настройка вывод. Тут всё просто. По схеме находим на какой вывод выведен ЦАП и настраиваем его как альтернативная функция.

```
GPIO_InitTypeDef GPIO_Init_DAC;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_Init_DAC.GPIO_Pin = GPIO_Pin_4;
GPIO_Init_DAC.GPIO_Mode = GPIO_Mode_AN;
GPIO_Init_DAC.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_Init_DAC.GPIO_OType = GPIO_OType_PP;
GPIO_Init_DAC.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOA, &GPIO_Init_DAC);
```

Теперь заполним ЦАП:

```
DAC_InitTypeDef          DAC_InitDAC;

RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
DAC_InitDAC.DAC_LFSRUnmask_TriangleAmplitude                          =
DAC_LFSRUnmask_Bit0;
DAC_InitDAC.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
DAC_InitDAC.DAC_Trigger = DAC_Trigger_None;
DAC_InitDAC.DAC_WaveGeneration = DAC_WaveGeneration_None;
```

```
DAC_Init(DAC_Channel_1, &DAC_InitDAC);

DAC_Cmd(DAC_Channel_1, ENABLE);
```

Разберём подробнее настройки.

Trigger – отвечает за автоматическое включение ЦАП. Можно настроить, что заданная величина будет устанавливаться по какому-то событию. В нашем случае данная опция отключено.

WaveGeneration – Генератор сигнала. Опять же не нужен. Отключается.

LFSRUnmask_TriangleAmplitude – отвечает за параметры генерируемого сигнала. Т.к. генерация сигнала не используется, то и эта опция тоже.

OutputBuffer – включает и выключает выходной буфер. Рекомендуется включить.

Чтобы установить нужное значение в ЦАП, надо использовать функцию:

```
DAC_SetChannel1Data(DAC_Align_12b_R, 0x0FFF);
```

Первым параметром задаётся выравнивание и разрядность. В данном случае это 12бит, выравненных по правому краю. Вторым параметром задаётся 16-тибитное значение, записываемое в этот ЦАП (из которых рабочими являются младшие 12 бит).

Работа с flash памятью

Чтение из Flash

Тут всё просто. У нас есть адрес из которого нам надо считать. Что мы и делаем функцией:

```
uint32_t flash_read(uint32_t address)
    {
    return (*(__IO uint32_t*) address);
    }
```

Эта функция возвращает данные из 4-х байт, начиная с заданного.

Разблокирование Flash

Прежде чем что-то делать с памятью, как уже было написано выше, её надо сначала разблокировать. Для этого надо в регистр KEYR сначала записать число 0x45670123, а затем 0xCDEF89AB. при использовании StdPeriph это сводится к вызову функции:

```
FLASH_Unlock();
```

Если вы закончили писать в память, то вызовите функцию:

```
FLASH_Lock();
```

Стирание Flash

Перед стиранием узнайте по приведённому выше списку в каком секторе находятся данные, подлежащие стиранию. Затем вызовите функцию:

```
FLASH_EraseSector(FLASH_Sector_5, VoltageRange_3);
```

FLASH_Sector_5 - Вместо 5 вставьте номер своего сектора.

VoltageRange_3 - это диапазон питания. Есть 4 диапазона:

VoltageRange_1 - питание от 1.8 до 2.1 В

VoltageRange_2 - питание от 2.1 до 2.7 В

VoltageRange_3 - питание от 2.7 до 3.6 В

VoltageRange_4 - питание от 2.7 до 3.6 В с внешним Vpp.

У вас скорее всего 3.3В без батарейки.

Запись в Flash

Запись также происходит по адресу одной из команд:

FLASH_ProgramDoubleWord() - запись 8-х байт.

FLASH_ProgramWord() - запись 4-х байт.

FLASH_ProgramHalfWord() - запись 2-х байт.

FLASH_ProgramByte() - запись 1-х байт.

Первым пишется адрес куда писать, вторым данные которые писать. Функции используются в зависимости от типа записываемых данных. Пример:

```
FLASH_ProgramWord(0x08007F00, 0x89ABCDEF);
```

Пример чтения и записи

Фунция которая записывает 12 символов в память:

```
#define DEVICE_ADDRESS 0x08007F00
#define DEVICE_SECTOR FLASH_Sector_1
void WriteDeviceAddress(char* data)
{
    uint8_t i;
    FLASH_Unlock();
    FLASH_EraseSector(DEVICE_SECTOR, VoltageRange_3);

for (i=0;i<12;i++)
{
  FLASH_ProgramByte(DEVICE_ADDRESS+i, data[i]);
}
FLASH_Lock();
}
```

Пример чтения записанных 12-ти символов:

```
void ReadDeviceAddress(char* Dout)
```

```
{
        uint32_t temp , i, j, k=0;
        for (i=0;i<3;i++)
        {
        temp = flash_read(DEVICE_ADDRESS+(i*4));
        for (j=0;j<4;j++)
        {
        Dout[k] = (char)((temp>>(j*8))&0xFF);
        k++;
        }
}
Dout[12]=0;
      }
```

Использование в программе:

```
      char TempStr[20], TempStr2[20];
      sprintf(TempStr2, "0001951337B1");
      WriteDeviceAddress(TempStr2);
      ReadDeviceAddress(TempStr);
```

Работа с памятью

При работе с памятью самое главное понять, что она устроена по принципу LittleEndian, т.е. меньший адрес имеет младший байт многобайтного числа. Поэтому, если число двухбайтное, то младший байт будет иметь тот же адрес, что и сама переменная, а старший – на единицу больше.

Например, возьмём такое число. Запишем в него значение 0xABCD. И посмотрим, как оно выглядит в памяти (рис. 2.23)

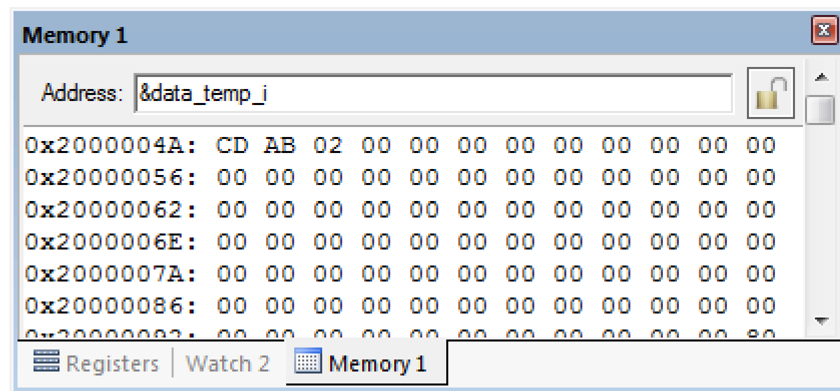

Рис. 2.23

Это может быть важно, к примеру, когда преобразуешь тип, присваивая 16-тибитный массив данных к 32-хбитному. Или получаешь 32-хбитное число из массива 4-х 8-ми битных чисел и т.д.

Программирование энкодера

Программирование кнопки

Схемы подключения кнопки рассмотрены в лабораторной 2. Для реализации схемы кнопки каждый из выводов подключается к одному из выводов микроконтроллера. Один из выводов настраивается как выход и устанавливается в 0. Это будет земля.

Второй вывод настраивается на вход и подтягивается к питанию. Теперь, если кнопка не нажата, на этом входе будет единица. Если же кнопка нажата, то на ней будет ноль, т.к. она окажется закороченной на землю.

Сам алгоритм программирования кнопки рассмотрен в лабораторной 2.

Программирование вращения энкодера

Как уже было сказано выше, в статичном состоянии на выходах A и B энкодера установлена логическая 1, т.к. оба вывода подтянуты к питанию. При вращении ножки A и B периодически подключаются к земле. В итоге на них появляются импульсы, показанные на рис 2.24.

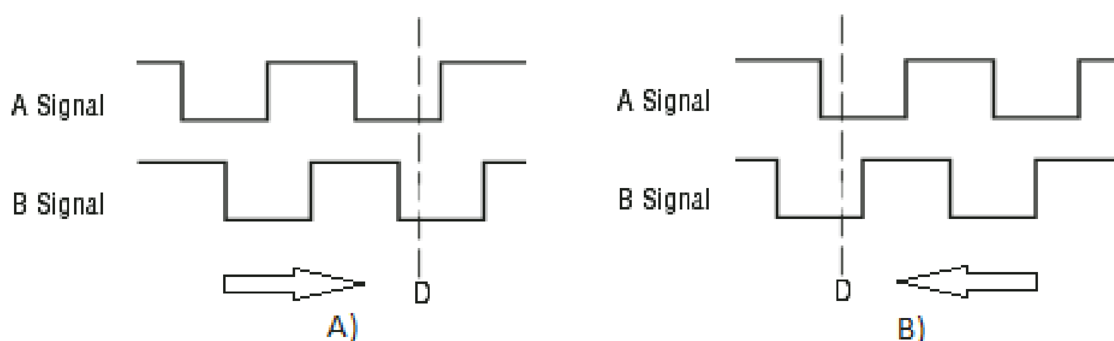

Рис 2.24

Как можно видеть, при вращении в одну сторону, сигнал A опережает сигнал B по фазе и на нём раньше устанавливается логический ноль. При вращении в другую сторону, наоборот, сигнал B опережает сигнал A по фазе. В итоге можно отметить, что на выходах A и B появляется так называемый код Грея, который при вращении в одну сторону выглядит так:

A: 1 -> 0 -> 1 -> 1 -> 1

B: 1 -> 1 -> 1 -> 0 -> 1

При вращении в другую сторону:

A: 1 -> 1 -> 1 -> 0 -> 1

B: 1 -> 0 -> 1 -> 1 -> 1

Кроме того, задача обработки усложняется тем, что у энкодера, как у любого другого механического устройства, присутствует дребезг.

Существует множество вариантов анализа получаемых с энкодера сигналов. Одним из которых является анализ того, какой из сигналов раньше установился в ноль. Сделать это можно следующим образом.

Во-первых, нужно избавиться от дребезга сигналов. Сделать это можно с помощью алгоритма борьбы с дребезгом для кнопки, рассматривая каждый вход как отдельную кнопку.

Далее можно рассматривать энкодер как идеальный.

Алгоритмов обработки сигналов в энкодера может быть множество. Рассмотрим алгоритм, в котором вывод о направлении поворота производится по тому, по какому выводу произошло изменение. Рассмотрим блок-схему алгоритма, представленную на рис 2.25

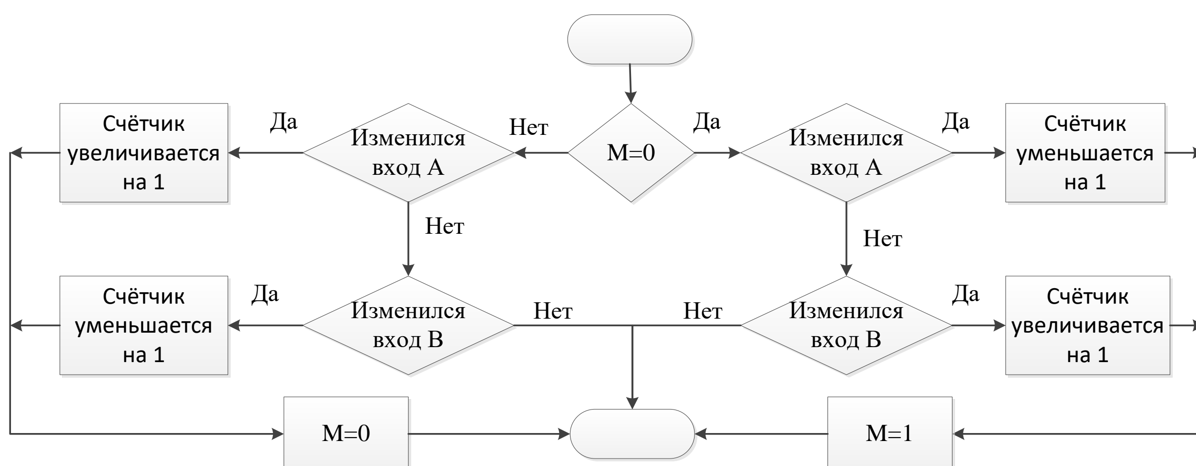

Рис. 2.25

Данный алгоритм вызывается периодически либо по таймеру, либо в основном цикле.

Можно заметить, что при вращении в одну сторону изменение состояния происходит у выводов по очереди. Если же начинается вращение в другую сторону, то изменение состояния произойдёт у одного и того же вывода. В данном алгоритме есть 2 состояния при M=0 и M=1. Если M=0, то считается, что изменение на выводе A – это вращение, к примеру, вправо, а изменение на выводе B – влево.

Отметим, что направление вращение не принципиально и может быть переопределено.

При изменении состояния одного из выводов, это отмечается в счётчике и меняется состояние на противоположное. И, для продолжения вращения в ту же сторону, ожидается изменение состояния уже на другом выводе.

Проверка изменения состояния производится путём сравнения текущего состояния и предыдущего. Если они не совпадают, то изменение было и предыдущему состоянию присваивается текущее.

Счётчик можно реализовать любым способом, а можно просто совершать действие, которое должно совершаться при вращении в нужную сторону. Стоит только помнить о том, что на один щелчок энкодера приходится 4 изменения состояния, т.е. рассчитанные значения будут превышать количество щелчков в 4 раза.

Рассмотрим работу алгоритма на примере (рис 2.26).

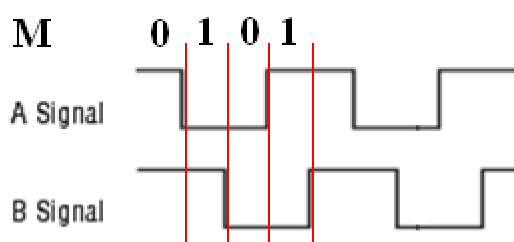

Рис 2.26

Пусть первоначально M=0 (самая левая часть сигнала на рис 2.24). Происходит изменение сигнала по линии A. Исходя из блок-схемы получается, что счётчик уменьшится, а состояние системы перейдёт в 1 (M

=1). Далее по линии B происходит изменение и, исходя из той же самой блок-схемы, счётчик опять уменьшится на 1, а M станет равным 0. Если теперь начать вращение в другую сторону, то опять произойдёт изменение по B и, согласно блок-схеме, счётчик увеличится на 1.

Принципы программирования клавиатуры

Предположим, что может быть нажата только одна кнопка. При этом учтём, что сложно нажимать кнопку с частотой выше 10 Гц. Т.е. на обработку клавиатуры должно уходить не более 100мс, чтобы это не было заметно для пользователя. При этом можно быть уверенным, что на протяжении хотя бы 50 мс нажатая кнопка будет оставаться нажатой.

Прежде всего производится сканирование всей клавиатуры. Для этого создаётся переменная, которая будет циклически принимать значения от 0 до 3.

```
uint8_t counter=0;
counter = (counter+1)%4;
```

Далее в зависимости от значения counter на соответствующую ножку выставляется 1. На остальных при этом выставляется 0. Считываются значения с входов клавиатуры. Если на одном из выводов оказывается единица, номер этого вывода запоминается и выдаётся сообщение о нажатии.

```
if (counter == 0){
  output0=1;
  output1=0;
  output2=0;
  output3=0;
} else if (counter == 1) {
  output0=0;
  output1=1;
  output2=0;
  output3=0;
} else if (counter == 2) {
  output0=0;
  output1=0;
  output2=1;
```

```
  output3=0;
 } else if (counter == 3) {
  output0=0;
  output1=0;
  output2=0;
  output3=1;
 }
counter = (counter+1)%4;
```

Считывать значение лучше перед изменением столбца. Это позволит пройти переходным процессам, если таковые будут иметь место.

```
if (counter == 0){
  if (input0 == 1) {
          res_keyboard = 1;
  } else if (input1 == 1) {
          res_keyboard = 2;
  } else if (input2 == 1) {
          res_keyboard = 3;
  } else if (input3 == 1) {
          res_keyboard = 4;
  } else {
          res_keyboard = 0;
  }
} else if (counter == 1) {
  if (input0 == 1) {
          res_keyboard = 5;
  } else if (input1 == 1) {
          res_keyboard = 6;
  } else if (input2 == 1) {
          res_keyboard = 7;
```

```
    } else if (input3 == 1) {
            res_keyboard = 8;
    }
  } else if (counter == 2) {
    if (input0 == 1) {
            res_keyboard = 9;
    } else if (input1 == 1) {
            res_keyboard = 0;
    } else if (input2 == 1) {
            res_keyboard = 11;
    } else if (input3 == 1) {
            res_keyboard = 12;
    }
  } else if (counter == 3) {
    if (input0 == 1) {
            res_keyboard = 13;
    } else if (input1 == 1) {
            res_keyboard = 14;
    } else if (input2 == 1) {
            res_keyboard = 15;
    } else if (input3 == 1) {
            res_keyboard = 16;
    }
    return res_keyboard;
  }
```

Обратите внимание, что res_keyboard обнуляется в начале, а возвращается в конце цикла сканирования.

Затем проверяется равна ли нулю возвращённая переменная. Если не равна, то счётчик, отвечающий за дребезг растёт, иначе уменьшается (если он больше нуля). Отсюда делается вывод нажата ли вообще кнопка. Если

нажата, то значение переменной res_keyboard показывает какая именно кнопка нажата. При этом, если кнопка нажата res_keyboard_filter = res_keyboard; Если отжата, то res_keyboard_filter = 0;

В бесконечном цикле уже проверяется какая кнопка была нажата.

```
uint8_t keyboard_press=0;

if (res_keyboard_filter == 1) {
  if (keyboard_press == 0)
  {
        keyboard_press == 1;
  //  Действие при нажатии
  }
} else if (res_keyboard_filter == 2) {

} else if (res_keyboard_filter == 3) {
…
} else if (res_keyboard_filter == 16) {

} else {
  if (keyboard_press == 1)
  {
        keyboard_press = 0;
        // Действие при отжатии.
  }
}
```

Переменная keyboard_press показывает была ли нажата кнопка. Переход keyboard_press из 0 в единицу показывает, что кнопка нажата, а переход из 1 в 0, что отжата.

Настройка генератора случайных чисел.

Функции работы с генератором случайных чисел (Random Number Generator) находятся в файлах stm32f4xx_rng.c и stm32f4xx_rng.h.

Тактирование включается функцией:

RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);

Включается генератор функцией

RNG_Cmd(ENABLE);

Считываются случайные значения функцией:

uint32_t RNG_GetRandomNumber(void);

Где вместо uint32_t подставляется переменная соответственного типа. Например:

Rand=RNG_GetRandomNumber();

Данная функция возвращает 32-х разрядное число. Т.е. число меняется от 0 до 2^32.

Подключение библиотеки для работы с SDIO с поддержкой fatfs

Содержимое архива помещается в отдельную папку внутри папки с проектом. В проекте создаётся отдельная группа и в эту группу добавляются все .c файлы из этой библиотеки. В итоге должно получиться, как показано на рис. 2.27.

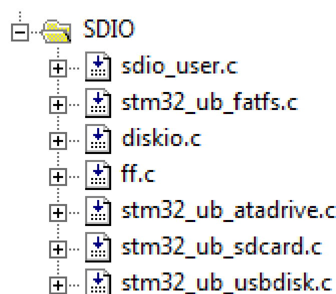

Рис 2.27

Затем следует прописать пути к используемым папкам. В итоге получится следующее (Рис. 2.28):

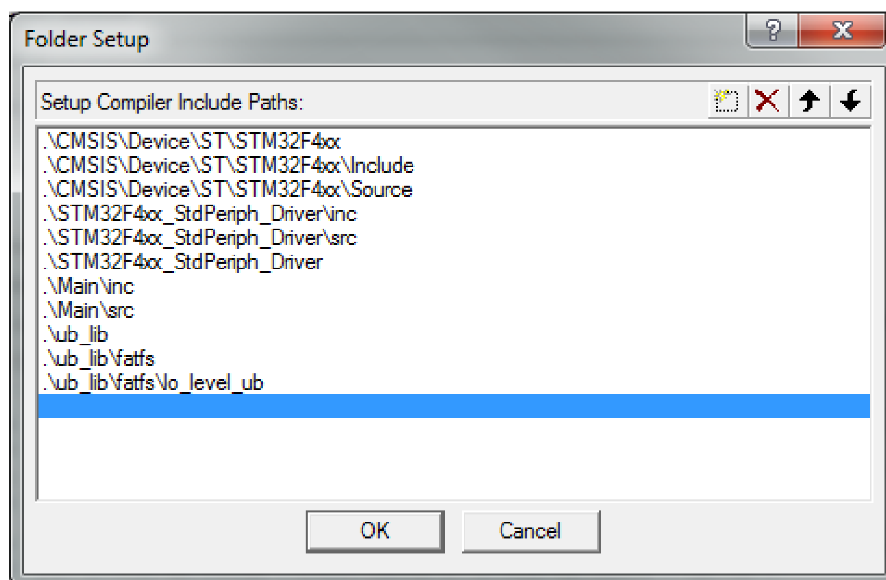

Рис. 2.28

Для подключения библиотеки к проекту, в main.h прописывается:

#include "stm32_ub_fatfs.h"

```
#include "sdio_user.h"
```

Для установки вывода выбора найдите строки:

```
//--------------------------------------------------------------
// Detect-Pin
// aktivieren oder deaktivieren
//    1 = mit Detect-Pin
//    0 = ohne Detect-Pin
//--------------------------------------------------------------
#define   USE_DETECT_PIN      1  // (mit Detect-Pin)
```

```
//--------------------------------------------------------------
// Detect-Pin der SD-Karte
//--------------------------------------------------------------
#if USE_DETECT_PIN==1
  #define SD_DETECT_PIN                  GPIO_Pin_13
  #define SD_DETECT_GPIO_PORT            GPIOH
  #define                                SD_DETECT_GPIO_CLK
RCC_AHB1Periph_GPIOH
#endif
```

В файле stm32_ub_sdcard.h. На схеме необходимо найти куда подключен вывод NCD и прописать его.

Для инициализации карточки используется строка:

filecheck = SDIO_FatFS_ini();

Если инициализация прошла успешно, то функция вернёт FATFS_OK.

Функции, используемые библиотекой, перечислены в файле stm32_ub_fatft.h в строках 64-74.

Ниже приведён небольшой пример записи строки в файл:

```
#include "stdio.h"
#include "stm32_ub_fatfs.h"
#include "sdio_user.h"

void main(void)
{
        FIL myFile;
        FATFS_t result;

        // Инициализация SDIO
        filecheck = SDIO_FatFS_ini();

        // Попытка создания файла 1.txt
        result = UB_Fatfs_OpenFile(&myFile, "0:/1.txt", F_WR_NEW);
```

```
	if (result != FATFS_OK) // если не получилось
	{
		BLUE_ON(); // вкл. Синий светодиод
		while(1){}
	} else { // Если файл создан
		YELLOW_ON();
	}
	UB_Fatfs_WriteString(&myFile, "Test string"); // Записываем строку в файл

	UB_Fatfs_CloseFile(&myFile); // Закрываем файл
	while(1)
	{
	}
}
```

Помните, что не закрыв файл или не синхронизировав запись, вы можете потерять записанные данные.

Часть 3: Полезные возможности языка Си

Использование функции sprintf

Функция **sprintf** включает в себя следующие параметры:

char *str – указатель на начало массива.

Str – строку, которую надо записать в массив.

Например, если надо записать в массив text слово Привет, надо использовать строку:

sprintf(text, “Привет”);

Если надо вывести число в массив, то это делается в зависимости от типа числа. Если число знаковое то используется директива %d. Например:

sprintf(text, “%d”, Rand); Тогда в text запишется значение переменной Rand.

Если же число беззнаковое, то используется %u. Например:

sprintf(text, “%u”, Rand); Тогда в text запишется значение переменной Rand но без знака.

Если надо ввести несколько данных в одну строку, то можно сделать так:

sprintf(text, “%u %u”, Rand[0], Rand[1]);

Использование функции atoi

Часто возникает необходимость преобразовать строку в число. Для этого можно либо это сделать самостоятельно, либо с использованием функции atoi.

Если это делать самостоятельно, следует учитывать, что символы кодируются от 0x30=0 до 0x39=9. Для перевода берётся число, потом число умножается на 10 и прибавляется следующее число и т.д.

Например, если buffer[0] = 0x31, buffer[0] = 0x35.

```
C=0;
I=0;
while (buffer[i] != 0)
{
        C=C*10;
        C=C+ buffer[I];
}
```

В результате получается C = 15.

Во втором случае преобразование будет выглядеть так:

```
C = atoi(buffer);
```

И опять C = 15.

Функция atoi преобразует число из строки до первой ошибки (числа из вне диапазона от 0x30 до 0x39).

Директива define

В языке Си есть возможность замены части кода неким текстом. Далее в программе, если использовать этот текст, компилятор заменит его соответствующим кодом.

Например, если в программе часто используется какая-то константа, такая как Пи, можно заменить число 3.14 на название константы строкой:

```
#define Pi 3.14
```

Теперь, можно использовать константу используя Pi вместо 3.14.

Но на это возможности директивы не заканчиваются. В неё можно записать любой код, заменив его на говорящее название. Например, пусть красный светодиод подключен к выводу D.14. Тогда включение светодиода можно записать:

```
#define RED_ON()                    GPIO_SetBits(GPIOD, GPIO_Pin_14)
```

Теперь в программе достаточно написать:

RED_ON();

И эта строка включит красный светодиод.

Такое использование позволяет упростить понимание кода. Кроме того, если светодиод будет подключен к другому выводу, то будет достаточно переписать новый вывод в директиве.

Аналогичным образом можно продолжить. Если заменить названием Порт и вывод в которым подключен светодиод, то получится:

```
#define RED_PORT                GPIOD
#define RED_PIN                 GPIO_Pin_14
#define RED_ON()                GPIO_SetBits(RED_PORT,
RED_PIN)
```

Названия RED_PORT и RED_PIN можно использовать в инициадизации порта. В итоге, если светодиод действительно будет подключен к другому выводу, то будет достаточно переписать эти два переименования и всё опять будет работать.

Можно создавать с помощью define и встраиваемые функции:

```
#define Add(a,b) a+b
```

Теперь, если написать:

```
uint8_t first = 10;
uint8_t second = 15;
uint8_t third;
third = Add(first, second);
```

Это будет аналогично:

```
third = first+second;
```

Создание нового файла

Если функций в программе много, то лучше всего их разбрасывать на разным файлам, сортируя по выполняемым функциям. Например, вынести в отдельный файл все функции работы со светодиодами.

Для этого в папке с проектом создаётся папка с названием общего назначения функций, которые будут там содержаться, например «Светодиоды». Далее создаются в этой папке 2 файла: .c и .h с одинаковыми названиями. Теперь нао прописать пути к этой папке и подключить .c файл к проекту. Открыв .c файл, пропишите:

```
#include “название_файла.h”
```

Так к исполняемому файлу будет подключен заголовочный. Щёлкните правой кнопкой мыши по этой строчке и выберите Open File. Откроется .h файл. В нём пропишите:

```
#ifndef НАЗВАНИЕ_ФАЙЛА_H
#define НАЗВАНИЕ_ФАЙЛА_H
#endif
```

В первой строке проверяется создан ли define. Если не создан, но компиляция файла выполняется. Если создан, но файл игнорируется. И несмотря на то, что .h файл может быть подключен многократно, выполняться его сборка будет только один раз.

После define надо прописать строку:

```
#include "stm32f4xx.h"
```

Эта строка подключит библиотеки микроконтроллера.

Теперь в .c файле можно создавать функции.

Создадим в .c файле функцию, складывающую 2 числа:

```
uint8_t add(uint8_t a, uint8_t b)
{
	return a+b;
}
```

Теперь в .h файле надо прописать прототип этой функции, чтобы она стала видна снаружи файла. Напишите внутри #ifndef:

```
uint8_t add(uint8_t a, uint8_t b);
```

Для использования этой библиотеки в .h файле другого модуля надо прописать:

#include “название_файла.h”

Это подключит библиотеку и позволит использовать все функции, прототипы которых прописаны в .h файле библиотеки.

Директивы define в библиотеке обычно пишутся в .h файле. В этом случае они оказываются доступны в других модулях.

Распределение такое: сначала идут define, затем прототипы функций.

В результате .h файл принимает вид:

```
#ifndef НАЗВАНИЕ_ФАЙЛА_H
#define НАЗВАНИЕ_ФАЙЛА_H
#include "stm32f4xx.h"
#define Pi 3.14
uint8_t add(uint8_t a, uint8_t b);
#endif
```

Часть 4: Алгоритмы

Алгоритм борьбы с дребезгом

При нажатии на кнопку в реальных условиях контакт происходит не мгновенно и происходит несколько замыканий-размыканий в результате которых несколько раз за единицы миллисекунд сменяются значения нуля и единицы на выходе кнопки. Из-за этого явления задача обнаружения нажатия кнопки становится не тривиальной.

Одним из алгоритмов борьбы с дребезгом является его программная фильтрация при которой программа ждёт когда сигнал примет устойчивое состояние. Идея заключается в том, чтобы, считывая состоние кнопки, увеличивать значение переменной на единицу каждый раз, когда это состояние принимает ожидаемое значение (например: кнопку нажали) и сбрасывать, если кнопка вернулась в исходное состояние. Если же значение кнопки достигло заданной величины, то считается, что мы получили устойчивое состояние нажатия и кнопка считается нажатой. Существует несколько реализаций этого алгоритма. Давайте реализуем один из них.

Прежде всего содаём переменную в которую будем записывать текущее состояние кнопки по мнению программы.

```
uint8_t button_state=1;//Кнопка отжата.
```

Сюда мы будем записывать готовое решение о нажатии на кнопку. Инициализируем значением того, что кнопка отжата. Напишем простейшую программу, изменяющую состояние светодиода по нажатию на кнопку. Но для начала определим 2 функции:

```
#define LED_INV() togglebit(GPIOD, GPIO_pin_14);
#define BUTTON_READ() read_bit(GPIOA,GPIO_pin_0);
```

Теперь напишем саму реализацию:

```
if (BUTTON_READ() == 0)//Кнопка нажата
{
        if (button_state == 1)
```

```
            {
                LED_INV();
  button_state = 0;
  }
}
else
{
            if (button_state == 0)
            {
  button_state = 1;
  }
}
```

В этой программе мы проверяем нажата ли кнопка. Если нажата (на ножке установлен 0), то мы проверяем переменную, в которую записываем состояние кнопки. Если состояние реальной кнопки а нашей переменной отличаются, то делается вывод, что это момент изменения состояния кнопки и светодиод меняет своё состояние. Сама же переменная принимает значение реальной кнопки.

Но эта программа никак не учитывает дребезг. Допишем переменную, которая будет уменьшаться до 0, если кнопка отжата и увеличиваться до заданного порога, если кнопка нажата.

```
uint8_t button_count=0;
if (BUTTON_READ() == 0)//Кнопка нажата
{
            if (button_count < 10) //счёт до порога
            {
                button_count++;
            } else if (button_state == 1) //Срабатывает если кнопка была отжата
            {
                LED_INV();
```

```
  button_state = 0;
   }
}
else // кнопка отжата
{
        if (button_count > 0) //счёт до 0
        {
                button_count--;
        }
        else if (button_state == 0) //Срабатывает если кнопка была нажата
        {
  button_state = 1;
   }
}
```

В написаной выше программе сначала счётчик досчитывает до заданного значения и только потом проверяет изменилось ли состояние кнопки.

Алгоритм организации задержки на Systick

Systick – это системный таймер, периодически выдающий прерывания через заданные промежутки времени. Это свойство таймера позволяет создавать задержки не загружая АЛУ. Рассмотрим для начала реализацию функции задержки на systick.

```
uint16_t delay_count=0;
void Systick_Handler(void)
{
        if (delay_count > 0)
        {
                delay_count--;
```

```
		}
	}

	void delayMS(uint16_t delay_data)
	{
		delay_count = delay_data;
		while(delay_count){};
	}
```

Аналогично можно реализовать задержку без функции delayMS. Для этого по окончанию отсчёта надо устанавливать флаг окончания ожидания. Причём устанавливать флаг в единицу надо когда уменьшаемая в Systick_Handler переменная равна 1, а не 0. Т.к., если устанавливать её при переменной равной нулю, флаг будет устанавливаться бесконечно. В итоге функция обработки прерывания выглядит примерно так:

```
	uint16_t delay_count=0;
	uint8_t delay_flag=0;
	void Systick_Handler(void)
	{
		if (delay_count > 1)
		{
			delay_count--;
		} if (delay_count == 1)
		{
			delay_flag = 1;
	  delay_count--;
		}
	}
```

Важно не забыть уменьшить ещё на 1 переменную, тогда когда она примет значение 1 и будет установлен флаг.

В функции main тогда мигание светодиодом будет выглядеть так:

```
int main(void)
{
  delay_count =501;

  while(1)
  {
          if (delay_flag == 1)
  {
          delay_flag = 0;
          LED_INV();
          delay_count =501;
  }
  }
}
```

Как видно в программе проверяется delay_flag. Если флаг равен 1, то выполняется следующие действия:

1) Сбрасывается флаг.
2) Инвертируется состояние светодиода.
3) Устанавливается новый отрезок времени записью значения отличного от 0 в переменную delay_count. Стоит заметить, что следует задавать значение на 1 большее, т.е. флаг устанавливается на 1, а не на 0.

Разумеется этот алгоритм имеет недостаток в виде не мгновенной реакции на установку флага. Причём, если программе зависнет в каком-либо цикле, флаг вообще не будет обработан. Потому, подобный алгоритм используется в частях кода не критичных к скорости работы. Если скорость реакции критична, то простейшая обработка производится прямо в прерывании. Кроме того, по установленному флагу можно заставить завершаться зависшие циклы. Так можно организовать timeout для циклов на тот случай, если есть шанс, что они будут выполнуться бесконечно.

Алгоритм TimeOut.

В USART бывают случаи когда приходит пакет не целиком. В это случае его необходимо сбросить и ожидать прихода следующего. Это делается по алгоритму, аналогичному тому, что описан ранее. По приёму байта данных устанавилвается переменная в какое-то значение, большее 1. Если приходят новые байты данных, то величина этой переменной обновляется. Обновление необходимо производить, если не предпологается приход всего пакета за заданный отрезок времени. Когда принят последний байт пакета данных, переменная программно сбрасывается в ноль и сброса пакета по Timeout (превышению времени ожидания) не происходит, т.к. флаг устанавливается при значении равном 1.

Если же часть байт задержались или вовсе не пришли, переменная доходит до единицы и производится сброс пакета данных путём обнуления счётчика пришедших данных, очищения буфера приёма и т.д. В итоге программа начинает ожидать следующий пакет данных.

Например:

```
uint8_t USART_data[256];
uint8_t USART_data_write=0;
uint8_t USART_data_read=0;
uint8_t USART_timeout_count=0;
uint8_t USART_timeout_flag=0;


//--------------------------------
void USART_Handler(void)
{
        USART_data[USART_data_write] = USART_Read_Data(USART3);
        USART_data_write++;
        if (USART_data_write == 20)
```

```
		{
			USART_timeout_count = 0;//выключили
		} else {
			USART_timeout_count = 200;//200 мс
  }

}

//--------------------------------
void Systick_Handler(void)
{
		if (USART_timeout_count > 1)
		{
			USART_timeout_count --;
		} else if (USART_timeout_count == 1)
		{
			USART_timeout_count --;
  USART_timeout_flag = 1;
		}
}

//--------------------------------
int main(void)
{
  while(1)
  {
		if (USART_timeout_flag == 1)
  {
		USART_data_write = 0;
  }
```

```
    }
}
```

Из этой программы видно, что в функции USART_Handler принимаются данные. При этом каждый раз обновляется счётчик для времени ожидания. При принятии же 20-го байта (Предполагается, что в пакете 20 байт), счётчик сбрасывается в 0.

Если же счётчик примет значение равное 1, установится флаг превышения ожидания и в функции main указатель на место записи обнулится.

Конечно, можно вместо выставления флага прямо в прерывании сбрасывать в 0 указатель на место записи. Это тоже не будет ошибкой, если вам не требуется ещё где-то обрабатывать данную ошибку.

Отправка строки с текстом по USART

Для передачи строки данных можно использовать процедуру:

```
void SendString_to_ USART (const char *str)
{
  while(*str != '\0')
  {
      while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
      USART_SendData(USART1, *str++);
  }
}
```

Конечный автомат

Зачастую вся программа или её часть имеет набор состояний, между которыми она должна переключаться по неким условиям. Самый простой пример – это светофор. Он имеет 5 состояний: Горит красный, горит жёлтокрасный, горит зелёный, мигает зелёный, горит жёлтый. И между этими режимами требуется переключаться. Граф светофора без учёта мигания зелёного приведён на рис. 2.29. Из рисунка можно видеть, что все состояния переключаются по прошествии некоторого времени, но условия переключения могут быть любыми.

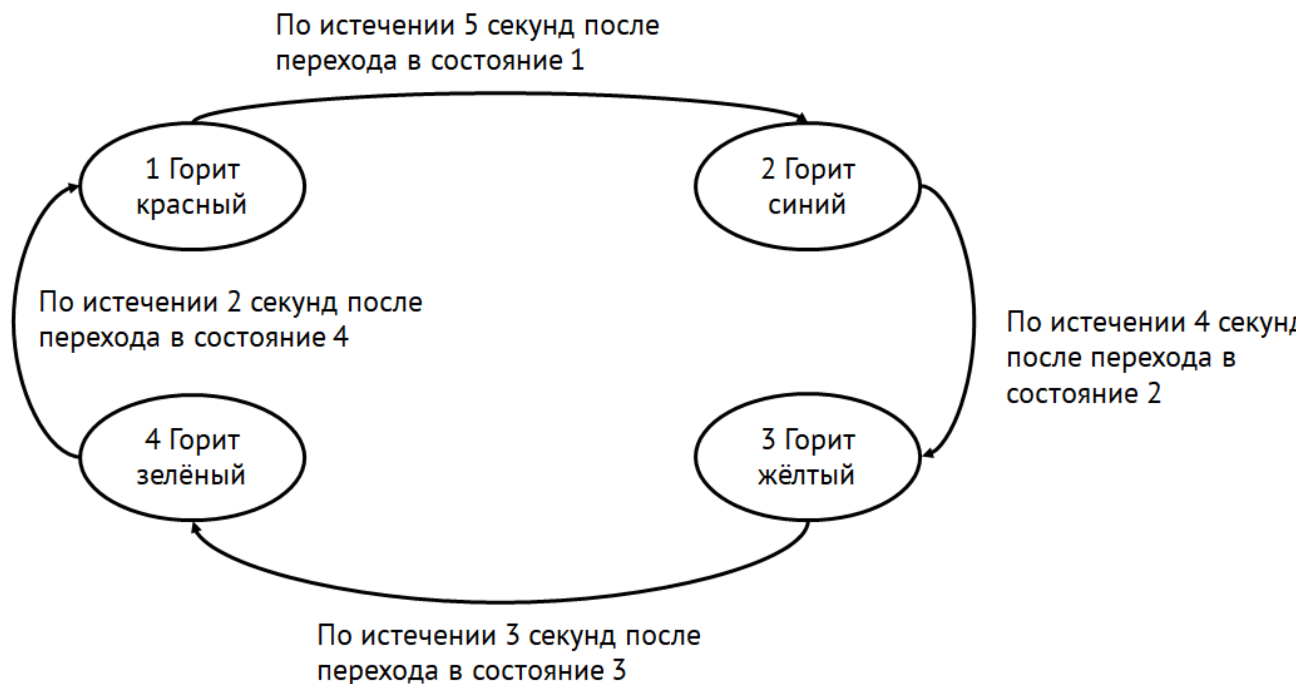

Рис. 2.29

Для организации такой структуры создаётся переменная состояния. Например, Mode. Затем записывается:

```
if (Mode == 0)
{
// Состояние 0
} else {
// Сотояние 1
} //и т.д.
```

В результате программа постоянно проверяет эту переменную и заходит в текущее состояние, где выполняет требуемые действия. В том числе и изменение состояние. В этом случае при следующем прооде программа зайдёт в уже другое состояние.

В общем случае каждое состояние может включать в себя блок инициализации, тело состояния и деинициализацию. В инициализации выполняются действия, которые нужно сделать однократно при попадании в это состояние. Тело состояния включает в себя действия, которые должна постоянно выполнять программа, находясь в этом состоянии. И последний блок – деинициализация, в котором выполняются действия, которые необходимо сделать при завершении работы данного состояния, перед переходом в следующее. Блок схема состояния приведена на рис. 2.30.

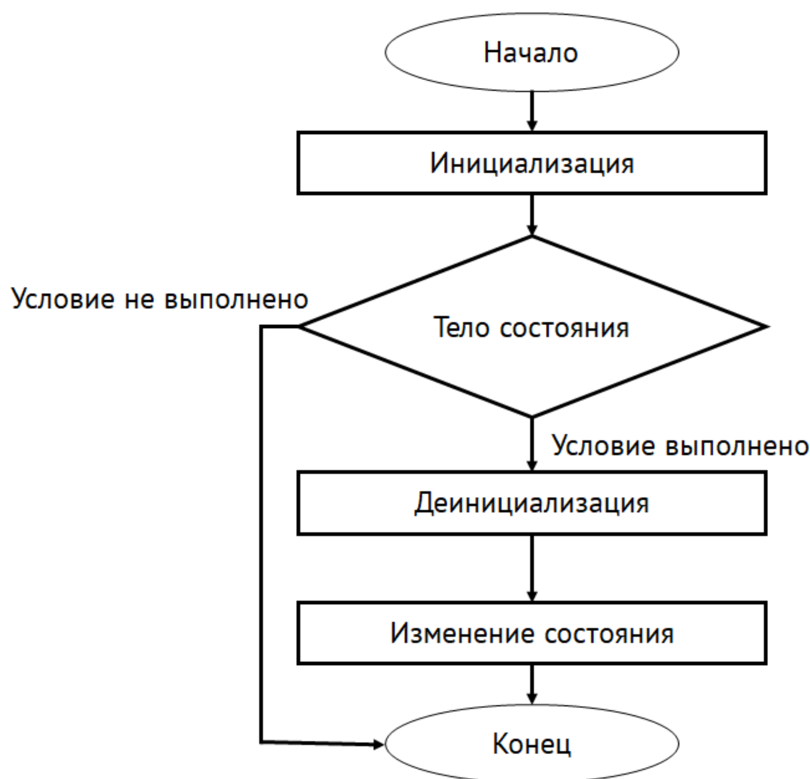

Рис. 2.30

При заходе в состояние программа проверяет выполнена ли инициализация. Если не выполнена, то выполняет, затем переходит к выполнению тела состояния. Здесь выполняются все требуемые действия и проверются все условия для перехода в другое состояние. Затем происходит

выход их состояния. Если же условие для перехода выполнено, то происходит деинициализация состояния и переключение на новое состояние.

Если состояния выполняются быстро, то в программе одновременно могут работать несколько конечных автоматов одновременно.

Рассмотрим работу конечного автомата на примере перемигивания двух светодиодов. Пусть красный горит 5000 мс, а зелёный – 3500 мс. Блок-схема состояния для одного светодиода представна на рис. 2.31.

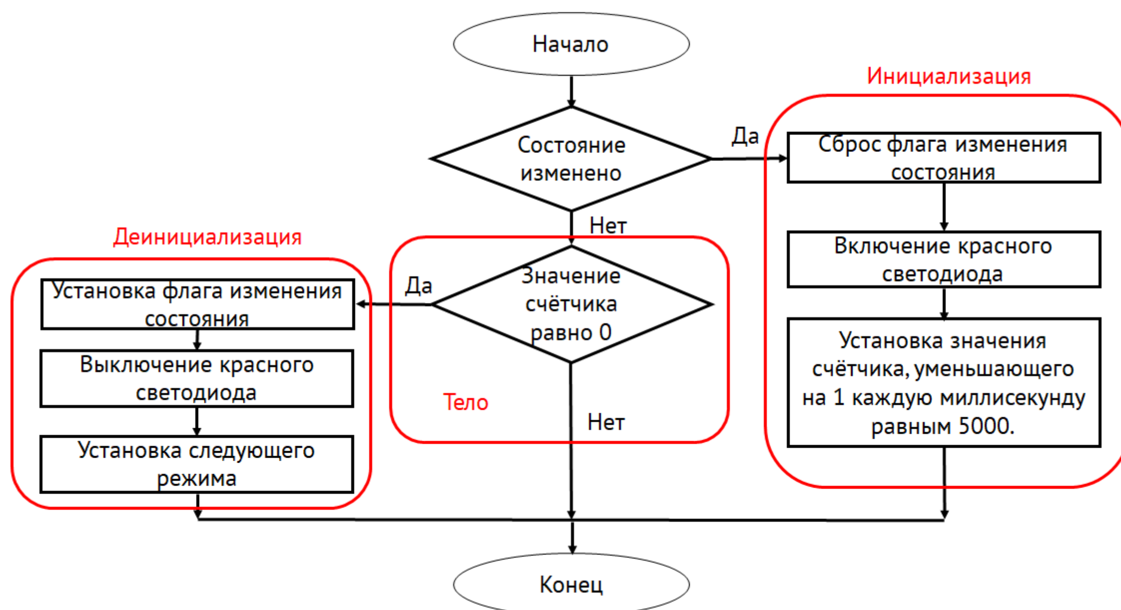

Рис. 2.31

В самом начале программа проверяет было ли изменено состояние и выполняет инициализацию состояния. В данном случае это

1) Сброс флага изменения состояния.
2) Включение красного светодиода. Он будет гореть всё состояние.
3) Установка значения счётчика на 5000 мс. Когда счётчик станет равным 0 -это условие перехода на следующее состояние.

В теле проверяется, стал ли счётчик равен 0. Если нет, то состояние завершает работу, передавая ресурсы МК другим частям программы. Если же счётчик оказался равным 0, то устанавливается новое состояние, выставляется флаг изменения состояния и выключается красный светодиод. Теперь при следующем проходе программа попадёт в состояние «горит зелёный».

Посмотрим, как это выглядит на практике.

Прежде всего определим переменную, отвечающую за режим работы МК.

```
uint8_t Mode=0;
```

Затем понадобится переменная, отвечающая за время задержки. В неё будет записываться время, которое должна выполняться та или иная часть программы. В системном таймере эта переменная будет уменьшаться до 0. В программе проверяется. Если переменная равна 0, значит время вышло.

```
uint16_t delay_count=0;

void Systick_Handler(void)
{
        if (delay_count > 0){
                delay_count--;
        }
}
```

При заходе в режим требуется его инициализировать, затем выполнить и в конце выключить всё, что больше не надо. Например:

```
if (Mode == 0)
{
  if (Start_Mode == 0)
  {
  GREEN_ON();
  Start_Mode = 1;
  delay_count = 3500;
```

```
    }
    if (delay_count == 0)
    {
GREEN_OFF();
Start_Mode = 0;
Mode = 1;// Включаем следующий режим.
    }
  }
```

Код выше включает светодиод, выставляет флаг, что инициализация прошла (Start_Mode = 1;). Выставляет счётчик на 3500 мс и ждёт, пока счётчик не обнулится. Потом выставляет новый режим работы и сбрасывает флаг инициализации режима.

Здесь всё, что под Start_Mode – это инициализация, а после delay_count == 0 – деинициализация. Сам if (delay_count == 0) – тело состояния.

Ниже представлен пример переключения зелёного и красного светодиодов.

```
uint8_t Mode=0;
uint8_t Start_Mode=0;
uint16_t delay_count=0;

void Systick_Handler(void)
{
        if (delay_count > 0){
```

```
			delay_count--;
		}
}

int main(void)
{
		while(1)
		{
  if (Mode == 0)
			{
				if (Start_Mode == 0)
				{
  GREEN_ON();
  Start_Mode = 1;
  delay_count = 3500;
  }
  if (delay_count == 0)
  {
  GREEN_OFF();
  Start_Mode = 0;
  Mode = 1;
  }

  } else if (Mode == 1)
			{
				if (Start_Mode == 0)
				{
  RED_ON();
  Start_Mode = 1;
  delay_count = 5000;
  }
  if (delay_count == 0)
  {
  RED_OFF();
  Start_Mode = 0;
```

```
Mode = 0;
}
                    }

}
}
```

Часть 5: Лабораторные

Как пользоваться таблицами вариантов

Задания в каждой лабораторной состоит из двух основных частей: общей, которую должны выполнить все и вариантной, индивидуальной для каждой бригады. Общий список заданий или варьируемых параметров представлен в первой таблице в задании к лабораторной работе. Во второй таблице задаётся соответствие вариантов и номеров заданий из первой таблицы.

Столбец «задание» показывает к какому конкретно заданию относится текущий варьируемый параметр.

Содержание отчётов

Отчёты должны быть выполнены согласно требованиям по оформлению отчётов и должны включать в себя следующие пункты:

1) Схема исследуемого лабораторного макета.
2) Блок-схемы программы.
3) Описание программы
4) Выводы по работе
5) Исходный код программы

Кроме перечисленных пунктов в отчёт помещается любая информация полученая в ходе выполнения лабораторной работы какая, как осциллограммы, фотографии работающего устройства, показания вольтметра, графики полученных данных.

Все результаты, полученные в ходе лабораторной работы помещаются в раздел результатов перед выводами. И потом на основе этих результатов делаются выводы.

Лабораторная 1: Использование библиотеки CMSIS

Задача

Написать программу, мигающую светодиодом с использованием библиотеки CMSIS.

Порядок выполнения работы

1) Создать пустой проект со своей фамилией в качестве названия, указав правильный микроконтроллер (см. «Создание проекта в keil uVission»).
2) Подключить библиотеку CMSIS (см. «Подключение библиотеки CMSIS»).
3) Произвести настройку библиотеки (см «Настройка библиотеки CMSIS»).
4) Убедиться, что проект полностью собирается без ошибок (см «Компиляция проекта»).
5) Настроить микроконтроллер на максимальную тактовую частоту (см. «Настройка тактирования микроконтроллера»).
6) Настроить отладчик микроконтроллера (см. «Программирование платы stm32f4-discovery»).
7) Записать программу в микроконтроллер (см. «Программирование микроконтроллера»).
8) Убедиться, что тактирование равно 168 МГц (см. «Определение тактовой частоты ядра»). Для этого надо воспользоваться выполнением программы по шагам и watch window.
9) Инициализировать вывод к которым подключен требуемый по заданию светодиод с использованием библиотеки CMSIS (см. «Настройка портов ввода-вывода с помощью библиотеки CMSIS»).
10) Написать программу, периодически устанавливающую инициализированный вывод то в логический ноль, то в логическую единицу с некоторой задержкой между этими событиями. Цикл на котором реализуется задержка берётся из таблицы. Из таблицы же берётся время, которое должна длиться задержка с точность 10%.

11) Для проверки времени задержки используется следующий алгоритм: Считаете сколько раз за заданный промежуток мигает светодиод. Делите одно на другое и ещё на два. В результате получится время, которое занимает задержка. Например, за 60 секунд светодиод мигнул 30 раз. Значит период мигания равен 2 секундам. Но в период мигания входит 2 задрежки. Поэтому время одной задержки равно 1 секунде.

Таблица 3.1: Список вариантов

Вариант	Светодиод	Цикл на котором основана задержка	Число миганий зелёного	Скорость работы выводов	Примерное время задержки
1	Красный	while	3	2 МГц	100 мс
2	Зелёный	for	5	50 МГц	250 мс
3	Жёлтый	while	3	50 МГц	500 мс
4	Красный	for	5	25 МГц	750 мс
5	Зелёный	while	4	2 МГц	1000 мс
6	Жёлтый	for	5	100 МГц	1250 мс
7	Красный	while	4	50 МГц	1500 мс
8	Зелёный	for	3	25 МГц	200 мс
9	Жёлтый	while	5	2 МГц	300 мс
10	Красный	for	4	50 МГц	400 мс
11	Зелёный	while	5	50 МГц	600 мс
12	Жёлтый	while	5	100 МГц	700 мс
13	Красный	for	3	2 МГц	800 мс
14	Зелёный	for	5	2 МГц	900 мс
15	Жёлтый	while	4	100 МГц	1100 мс
16	Красный	for	3	100 МГц	1200 мс

17	Зелёный	while	3	25 МГц	1300 мс
18	Жёлтый	for	4	25 МГц	1500 мс
19	Красный	while	4	25 МГц	330 мс
20	Зелёный	for	4	100 МГц	660 мс
21	Жёлтый	for	4	2 МГц	2000 мс
22	Красный	for	3	50 МГц	2500 мс
23	Зелёный	while	3	100 МГц	1750 мс

Лабораторная 2: Использование библиотеки StdPeriph

Задача

Написать программу, мигающую светодиодом с использованием библиотеки StdPeriph.

Порядок выполнения работы

1) Сделать копию проекта из лабораторной работы 1.
2) Добавить в проект билиотеку stdperiph (см. «Подключение библиотеки StdPeriph»).
3) Убедиться, что программа собирается.
4) Инициализировать выводы к оторым подключены светодиоды с использованием библиотеки StdPeriph (см. «Настройка портов ввода-вывода с помощью библиотеки StdPeriph»).
5) Написать программу работы светофора с помощью библиотеки StdPeriph в зависимости от варианта, номер которого совпадает с номером бригады (см. таблицу 3.1). При написании программы обязательно требуется вынести всю инициализацию порта в отдельную функцию в отдельном файле. А все выводы прописать в директиве define.

Содержание отчёта.

В отчёте нужно в том числе сделать выводы о практической разнице между исследованными библиотеками.

Лабораторная 3: Таймеры

Цель

Цель работы научиться использовать таймеры для реализации программ с привязкой к реальному времени.

Задачи

1) Написать, используя системный таймер, мигание светодиодами с временем горения и не горения согласно варианту (табл. 3.4).
2) Сгенерировать прямоугольный сигнал на заданном выводе путём изменения состояния вывода микроконтроллера с заданной частотой.
3) Получить осциллограммы сигнала, отправляемого на один из светодиодов и сгенерированного сигнала.
4) Задания выполняются без использования функции задержки.

Содержание отчёта:

В выводах в частности, сравнить 2 вида таймеров.

Таблица 3.4: Список вариантов

Вариант	Задание					
	LED1	T1 с	LED2	T2 с	f Гц	Вывод
1	Синий	0.5	Красный	1.3	1 000 000	PA3
2	Жёлтый	1	Зелёный	0.9	750 000	PA5
3	Синий	1.5	Красный	0.6	500 000	PA7
4	Жёлтый	0.5	Зелёный	1.65	1 500 000	PA9
5	Синий	1	Красный	2.1	100 000	PA1
6	Жёлтый	1.5	Зелёный	1.3	200 000	PE1
7	Синий	0.5	Красный	0.9	250 000	PA15
8	Жёлтый	1	Зелёный	0.6	1 250 000	PA1
9	Синий	1.5	Красный	1.65	125 000	PB1
10	Жёлтый	0.5	Зелёный	2.1	300 000	PB3
11	Синий	1	Красный	1.3	800 000	PB5

12	Жёлтый	1.5	Зелёный	0.9	1 100 000	PB7
13	Синий	0.5	Зелёный	0.6	550 000	PB9
14	Жёлтый	1	Красный	1.65	350 000	PB11
15	Синий	1.5	Зелёный	2.1	650 000	PB13
16	Жёлтый	0.5	Красный	1.3	850 000	PB15
17	Синий	1	Зелёный	0.9	450 000	PC1
18	Жёлтый	1.5	Красный	0.6	350 000	PC3
19	Синий	0.5	Зелёный	1.65	950 000	PC5
20	Жёлтый	1	Красный	2.1	900 000	PC7
21	Синий	1.5	Зелёный	0.9	333 000	PC9
22	Жёлтый	0.5	Красный	0.6	150 000	PC11
23	Синий	1	Зелёный	1.65	550 000	PC13
24	Жёлтый	1.5	Красный	2.1	400 000	PC15

LED1 – Светодиод 1.

T1 – Время горения (не горения) светодиода 1.

LED2 – Светодиод 2.

T2 – Время горения (не горения) светодиода 2.

f Гц – Частота, изменения состояния вывода.

Вывод – Название вывода, состояние которого требуется менять.

Лабораторная 4: Порты общего назначения в режиме входа

Задачи

1) Найти кнопку на схеме.

2) Инициализировать кнопку.

3) Реализовать включение (или выключение светодиода в зависимости от варианта) светодиода при зажатой кнопке.

4) Написать программу изменения состояния светодиода в момент нажатия (отжатия) на кнопку.

5) Снять осциллограмму сигнала с кнопки.

6) Задания выполняются без использования функции задержки.

Содержание отчёта:

Обязательно наличие осциллограмм дребезга.

В выводах в частности, необходимо сравнить алгоритмы борьбы с дребезгом.

Таблица 3.5: Список вариантов

Номер	Параметр	Задание
1	При зажатой кнопке светодиод горит	3
2	При зажатой кнопке светодиод не горит	3
3	Скорость работы выводов: 2МГц	
4	Скорость работы выводов: 25МГц	
5	Скорость работы выводов: 50МГц	
6	Скорость работы выводов: 100МГц	
7	Красный	4
8	Синий	4
9	Зелёный	4
10	Жёлтый	4
11	Все	4

Таблица 3.6: Список вариантов

Вариант	Задание		
1	1	3	7
2	2	4	8
3	1	5	9
4	2	3	10
5	1	4	11
6	2	5	7
7	1	6	8
8	2	3	9
9	1	4	10
10	2	5	11
11	1	3	7
12	1	4	8
13	2	5	9
14	2	6	10
15	1	6	11
16	2	5	7
17	1	4	8
18	2	3	9
19	1	5	10
20	2	6	11
21	2	6	8
22	2	5	9
23	1	4	10
24	1	3	11

Лабораторная 5: USART

Задание

Реализовать протокол в зависимости от варианта (табл. 3.7, 3.8, 3.9).

Первые 3 задания выполняются в одной программе. Четвёртое – в другой.

Задания берутся из соответствующей используемому протоколу таблицы.

По-умолчанию светодиоды мигают. Время горения – 500 мс. Время не горения – 500 мс.

Комментарии к заданиям приведены после таблиц.

Примечание: Символьный протокол использует символы для передачи данных. В результате пакет представляет собой некую строку. А число 0 –её окончание.

Бинарный протокол осуществляет передачу данных числами.

Таблица 3.7

№	Скорость	Протокол				
1	2400	Символьный	1	2	4	1
2	4800	Бинарный	1	5	6	2
3	9600	Символьный	1	3	7	3
4	19200	Бинарный	1	2	12	1
5	38400	Символьный	1	5	8	2
6	57600	Бинарный	1	3	9	3
7	115200	Символьный	1	6	12	1
8	2400	Бинарный	1	4	11	2
9	4800	Символьный	1	2	10	3
10	9600	Бинарный	1	7	13	1
11	19200	Символьный	1	3	11	2
12	38400	Бинарный	1	2	8	3
13	57600	Символьный	1	4	13	1

14	115200	Бинарный	1	3	7	2
15	2400	Символьный	1	5	9	3
16	4800	Бинарный	1	4	12	1
17	9600	Символьный	1	6	12	2
18	19200	Бинарный	1	8	13	3
19	38400	Символьный	1	5	11	1
20	57600	Бинарный	1	9	10	2
21	115200	Символьный	1	2	10	3
22	9600	Бинарный	1	2	11	1

Таблица: 3.8

№	Команда для бинарного протокола
1	Команда запроса ID устройства. Возвращает фиксированное число в блоке данных.
2	Задать время горения светодиодов. В поле данных на нечётных местах идут номера светодиодов, на чётных – соответствущее время горения светодиода при мигании.
3	Задать время негорения светодиодов. В поле данных на нечётных местах идут номера светодиодов, на чётных – соответствущее время горения светодиода при мигании.
4	Задать число миганий светодиодов. В поле данных на нечётных местах идут номера светодиодов, на чётных – соответствущее количество миганий. После получения этой команды светодиоды мигают заданное число раз и выключаются.
5	Задать последовательность включений светодиодов. В поле данных записываются номера светодиодов, которые должны включаться. Время горения каждого светодиода определяется командой 6. По-умолчанию: 1 секунда. После получения этой команды светодиоды перестают мигать и выполняется полученная команда.
6	Задать время горения каждого светодиода из задания 5. Это время

	одно для всех.
7	Преобразовать число из десятичной системы счисления в двоичную. С ПК отправляется число от 0 до 255, надо вернуть преобразованное в виде восьми байт, содержащих только 0 или 1.
8	Преобразвовать число из двоичной системы счисления в десятичную. С ПК отправляется 8 байт, содержащих только 0 или 1, надо вернуть преобразованное в беззнаковое целое.
9	Калькулятор. В первом байте отправляется число, во втором код оператора, в третьем – второе число. Требуется вернуть результат.
10	Запрашивает состояние кнопки и светодиодов.
11	Включает-выключает мигание светодиодов. В одном пакете можно вменить режим работы нескольких светодиодов.
12	Контроль ошибок в пакете (CRC).
13	Переключает режим работы светодиодов с мигания на переключение по-кругу и обратно. В поле данных передаётся время горения светодиодов как во время движения по-кругу, так и в процессе мигания. Для каждого светодиода отдельно.

Таблица: 3.9

№	Команды для символьного протокола
1	Команда запроса ID устройства. Возвращает фиксированное число в блоке данных.
2	Задать время горения светодиода в мс при мигании. В поле данных записывается число от 100 до любого. Если число меньше 100, то команда игнорируется. Задаётся отдельная команда для каждого светодиода.

3	Задать время негорения светодиода в мс при мигании. В поле данных записывается число от 100 до любого. Если число меньше 100, то команда игнорируется. Задаётся отдельная команда для каждого светодиода.
4	Запрашивает состояние кнопки и светодиодов.
5	Включает-выключает мигание светодиодов. В одном пакете можно вменить режим работы нескольких светодиодов.
6	Преобразовать число из десятичной системы счисления в двоичную. С ПК отправляется число в поле данных в виде строки от 0 до 255, надо вернуть преобразованное.
7	Преобразовать число из двоичной системы счисления в десятичную. С ПК отправляется число в поле данных в виде строки xxxxxxxx, надо вернуть преобразованное. Где x либо 0, либо 1.
8	Преобразовать число из десятичной системы счисления в шестнадцатеричную. С ПК отправляется число в поле данных в виде строки от 0 до 255, надо вернуть преобразованное.
9	Преобразовать число из шестнадцатеричной системы счисления в десятичную. С ПК отправляется число в виде строки в шестнадцатеричном виде от 0 до FF, надо вернуть преобразованное.
10	Контроль ошибок в пакете (CRC).
11	Реализовать калькулятор. С ПК отправляется строка x+y. Надо вернуть результат. Числа от 0 до 9.
12	Задать число миганий светодиодов. В поле данных отправляется числа, соответстсующих 4-м светодиодам. Если не надо, чтобы светодиод мигал, отправляется символ нуля.
13	Преобразование типов. С ПК отправляется в поле данных беззнаковое однобайтное число. Требуется вернуть соответствующее ему число в знаковое.

Таблица 3.10	
№	Команда для второй части задания.
1	При работе светофора с компьютера передаётся команда с указанием в какое состояние должен перейти светофор. В поле данных указывается номер состояния светофора. Например, с ПК может прийти указание переключиться в состояние горящий зелёный (всего 4 состояния – красный, красный + желтый, желтый зеленый).
2	Команда, которая при работающем светофоре переводит его в режим выключено (мигающий желтый). Если светофор уже в этом режиме, то он из него выходит в нормальный режим работы.
3	Запрашивает текущее состояние светофора.

Порядок выполнения работы

1) Выбирается номер UART, который планируется использовать.
2) Выбираются выводы для TX и RX для UART. При этом нельзя выбирать вывод PA9, т.к. он подключен к USB и данные через этот вывод передаваться не будут.
3) Настроить UART на приём и передачу.
4) Написать программу, которая принимает с ПК 1 байт даных.
5) Подключить плату FTDI к ПК.
6) Запустить диспетчер устройств и найти номер COM порта, который соответствует плате FTDI. Для того, чтобы убедиться, что вы правы, вытащите и воткните плату ещё раз.

7) Если номер полученного COM порта меньше, чем 10, используейте программу ComMon, если более, то – Com Test.
8) Установить точку останова в функцию прерывания от UART.
9) Отправить 1 байт данных на МК.
10) Убедиться, что программа попала в эту функцию.
11) Принять байт и проверить, что он совпадает с отправленным.
12) Пока эти пункты выполнены не будут, дальше двигаться нет смысла.
13) Когда приём будет отлажен, написать программу, отправляющую 1 байт через заданный промежуток времени. Этот байт должен совпадать с тем, что был в допуске.
14) Принять на ПК этот байт и убедиться, что он принимается правильно.
15) С помощью осциллографа получить изображение этого сигнала.
16) Сфотографировать его и сохранить для отчёта.
17) Когда приём и передача отлажены, можно приступать к написанию программы.
18) Написать в функции прерывания функцию приёма пакета в зависимости от требуемого протокола. Для бинарного: «Протокол с данными переменной длины». Для символьного протокола: «Протокол со стоп байтом».
19) Убедиться, что пакеты правильно разбираются программой, в буфер попадают правильные данные и выставляется флаг окончания приёма данных.
20) Написать программу, которая мигает светодиодами при включении. Время мигания светодиодов задаётся в переменных.
21) Написать функцию, которая проверяет флаг приёма. Если он равен 1, то сбрасывает его и начинает выполнение принятого пакета.
22) Написать внутри этого if проверку команд оператором case или набором операторов if.

```
if (command == 0)
{
```

```
        //Выполняется команда 0
} else if (command == 1)
{
        //Выполняется команда 1
}
```

23) Написать выполнение команд.
24) Реализовать таймаут для сброса частично пришедших пакетов.

Возможные ошибки и их решения

1) COM порт не появлется.

   Скорее всего не стоят драйверы для платы. Попробуйте воткнуть в соседний порт. Часто драйверы ставятся только на один из портов. Если так и не появился, зовите преподавателя.

2) Байт не принимается. Программа не попадает в функцию прерывания.

   Если порт определился, но программа не попадает в функцию прерывания, убедитесь, что светодиод мигает при передаче байта, т.к. у большинства плат стоят светодиоды, мигающие при передаче данных.

   Если светодиод не мигает, перезапустите программу работы с портом, т.к. скорее всего она просто потеряла связь с портом. Или вернитесь к вопросу 1.

   Если светодиод мигает, то проверьте подключение платы к микроконтроллеру. Помните, что RX подключается к TX и наоборот. Подключите только TX платы к RX микроконтроллера, чтобы не перепутать.

   Далее проверьте в настройке тактирования, что функция включения тактирования соответствует передаваемому значения. Функция должна быть:

   RCC_APB1PeriphClockCmd

   И содержать APB1 или APB2 в зависимости от подключаемого UART.

   Затем проверьте по шагам, что функция настройки выполняется и вы включили тактирование порта к подключаете UART, настроили выводы на

AF, Включили тактирование самого UART, настроили UART на передачу и приём. И скорость передачи тоже верная. А сам UART вы не забыли включить. И включены разрешения прерываний от UART и от приёма UART.

Если всё проверено, но не работает, обращайтесь к преподавателю.

3) Байт данных не передаётся на ПК.

   Решение аналогичное предыдущему.

4) Данные принимаются не те, что передаются.

   Проверьте правильно ли настроена частота передачи. Она должна на ПК и МК совпадать.

   Если не помогло, то проверьте настройку тактирования МК. В частности HSE должно быть равно 8 000 000.

   Если не помогло, используйте осциллограф для того, чтобы посмотреть реальную частоту сигнала.

Содержание отчёта:

Указать на осциллограмме части пакета (Стоп бит, старт бит, старший бит, младший бит).

Дать описание протокола. В это описание входит перечень всех используемых команд и формат данных для каждой команды. Предусматривает ли команда наличие данных. Если да, то для чего данные используются и в каком формате эти данные передаются.

Лабораторная 6: LCD

Задачи

1) Вывести счётчик на дисплей.
2) Вывести слово Hello (или «Привет»).
3) Вывести строку, переданную с ПК по USART.

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
6) Отчёт может включать дополнительные пункты.

Таблица 3.11: Список вариантов

Номер	Параметр	Задание
1	Вывести строку на первой строке дисплея	
2	Вывести строку на второй строке дисплея	
3	Выравнивание левому краю	
4	Выравнивание правому краю	
5	Выравнивание центру	
6	Русский текст	
7	Английский текст	
8	Выводы: PA1-11	
9	Выводы: PB0-10	
10	Выводы: PC0-10	
11	Выводы: PD0-10	
12	Выводы: PB5-15	
13	Выводы: PC5-15	
14	Выводы: PD5-15	

Варианты

Таблица 3.12

№	Задание			
1	1	3	6	8
2	2	4	6	9
3	1	5	6	10
4	2	3	6	11
5	1	4	6	12
6	2	5	6	13
7	1	3	6	14
8	2	4	6	8
9	1	5	6	9
10	2	3	6	10
11	1	4	6	11
12	2	5	6	12
13	1	3	7	13
14	2	4	7	14
15	1	5	7	8
16	2	3	7	9
17	1	4	7	10
18	2	5	7	11

19	1	3	7	12
20	2	4	7	13
21	1	5	7	14
22	2	3	7	8
23	1	4	7	9
24	2	5	7	10

Лабораторная 7: ADC

Задачи

1) Определиться с номером АЦП.
2) Настроить АЦП.
3) Снять значение с АЦП при замыкании вывода на 3 В. Передать на ПК 2 варианта и непосредственное значение снятое с АЦП и пересчитанное в напряжение.
4) Снять значение с АЦП при замыкании вывода на 0 В. Передать на ПК 2 варианта и непосредственное значение снятое с АЦП и пересчитанное в напряжение.
5) Снять значение с термодатчика. На ПК передать температуру с точностью до 10-х градуса.
6) Снять напряжение с никуда не подключенной ножки. Передать на ПК 2 варианта и непосредственное значение снятое с АЦП и пересчитанное в напряжение.
7) Снять осциллограмму напряжения с ножки, с которой снимается сигнал в задании 6.

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
5) Выводы.
6) Отчёт может включать дополнительные пункты.

Таблица 3.13: Список вариантов

Номер	Частота дискретизации	АЦП
1	1 Гц	1
2	5 Гц	2
3	10 Гц	3
4	25 Гц	1
5	50 Гц	2
6	75 Гц	3
7	100 Гц	1
8	200 Гц	2
9	250 Гц	3
10	300 Гц	1
11	400 Гц	2
12	450 Гц	3
13	500 Гц	1
14	600 Гц	2
15	700 Гц	3
16	750 Гц	1
17	800 Гц	2
18	900 Гц	3

19	1000 Гц	1
20	1100 Гц	2
21	1200 Гц	3
22	1250 Гц	1
23	1300 Гц	2
24	1400 Гц	3
25	1500 Гц	1

Лабораторная 8: Память

Задачи

1) Стереть сектор, в который планируется запись.
2) Считать значение по заданному адресу.
3) Записать не нулевое значение, считать его.
4) Записать другое значение в ту же ячейку, не стирая оной.
5) Считать его.
6) Сравнить с записанными ранее. Стереть сектор. Записать последнюю цифру заново.
7) Считать её.
8) Написать программу счётчика включений платы после программирования.

Содержание отчёта:

1) Исследуемая схема.

2) Блок-схемы программ.

3) Описание программ.

4) Программы.

5)

6) Выводы. В частности, сделать выводы об особенностях работы памяти NAND.

7) Отчёт может включать дополнительные пункты.

Лабораторная 9: SPI

Задачи

1) Настроить SPI.
2) Написать программу зажигающую светодиод в зависимости от наклона.
3) Отправлять снятые данные с акселерометра по заданному протоколу.
4) Выполнить задание в зависимости от варианта.
5) Снять осциллограмму работы SPI (clk + MOSI) при передаче 1 байта.

№	Задание
1	Счётчик ударов по столу рядом с платой. Результат отправляется по UART на ПК.
2	Когда телефон (на вибрации) не звонит горит красный светодиод, когда звонит – зелёный.
3	Счётчик ударов по плате при игнорировании по столу рядом.
4	Счётчик редких ударов. Повторные удары в течении одной секунды игнорируются.
5	Если плата в покое – горит зелёный. Если плату понесли – красный. Удары игнорируются.
6	Плата по UART выдаёт информацию о своём положении (лежит горизонтально, стоит на торце и т.д.)
7	Двойной удар по плате или рядом переключает светодиод на следующий по порядку.
8	Встряска платы переключает светодиод на следующий по порядку.
9	Счётчик оборотов по каждой оси (по которым возможно). Результат отправляется по UART на ПК.
10	Написать программу, которая отправляет на ПК время 1 цикла вибрации телефона во время звонка в миллисекундах.

Содержание отчёта:

1) Исследуемая схема.

2) Блок-схемы программ.

3) Описание программ.

4) Программы.

5) Выводы.

6) Отчёт может включать дополнительные пункты.

Лабораторная 10: Энкодер

Задачи

1) Написать программную реализацию энкодера.
2) Написать аппаратную реализацию энкодера.
3) Увеличивать переменную при повороте энкодера в одну сторону и уменьшать переменную при повороте в другую. При изменении значения переменной отправлять её на ПК.
4) Снять осциллограмму работы энкодера.
5) Написать программу, которая переключает последовательно светодиоды по кругу.

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
5) Выводы. В частности, сделать выводы об особенностях обоих алгоритмов работы с энкодером.
6) Отчёт может включать дополнительные пункты.

Лабораторная 11: ЦАП

Задачи

1) Сгенерировать на выходе 3 вольта.
2) Сгенерировать на выходе 0 вольт.
3) Сделать постепенное зажигание и гашение светодиода. Вычислить диапазон напряжений в котором светодиод меняет свою яркость.
4) Период мигания светодиода и номер ЦАП указаны в задании (надо будет написать.)
5) Генерация сигнала из задания и снятие его осциллограммы. (пила, треугольник и синусоида)

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
5) Выводы.
6) Отчёт может включать дополнительные пункты.

Лабораторная 12: Клавиатура

Задачи

1) Зажигать комбинацию светодиодов в зависимости от нажатой кнопки.
2) Отправлять на ПК номер нажатой кнопки.

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
5) Выводы.
6) Отчёт может включать дополнительные пункты.

Лабораторная 13: Генератор случайных чисел

Задачи

1) Считать значения с генератора случайных чисел, передать их на USART. Не менее 200 значений. Сверху число не ограничено.
2) Найти распределение случайной величины, разбив весь диапазон на N частей и, сделав не менее 10000 экспериментов, посчитать количество попаданий в каждый диапазон. Количество попаданий в каждый поддиапазон вывести на ПК.

Содержание отчёта:

1) Исследуемая схема.

2) Блок-схемы программ.

3) Описание программ.

4) Программы.

5) График снятых с генератора значений.

6) Гистограмма полученного распределения.

7) Выводы. В частности, о применимости данного генератора как генератора случайных чисел.

8) Отчёт может включать дополнительные пункты.

Лабораторная 14: Динамическая индикация

Задачи

1) Написать программу, которая зажигает 1 светодиод в зависимости от количества включений платы (пересбросов микроконтроллера).

Содержание отчёта:

1) Исследуемая схема.
2) Блок-схемы программ.
3) Описание программ.
4) Программы.
5) Выводы.
6) Отчёт может включать дополнительные пункты.

Лабораторная 15: SD карта

Задачи

1)

Содержание отчёта:

1) Исследуемая схема.

2) Блок-схемы программ.

3) Описание программ.

4) Программы.

5) Выводы.

6) Отчёт может включать дополнительные пункты.