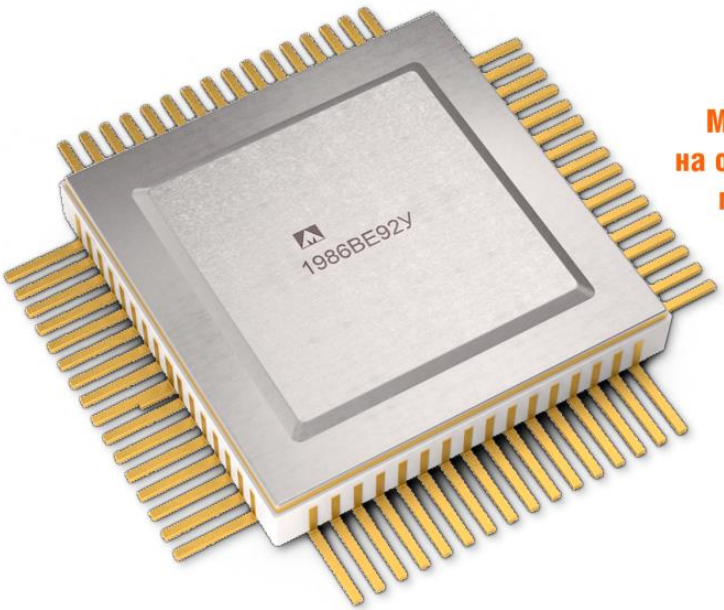
А.В. Благодаров
Л.Л. Владимиров

Программирование микроконтроллеров



Методическое пособие на основе отечественных микросхем семейства 1986BE9x разработки и производства компании «Миландр»

Москва
2016

Рассматриваются основы программирования на языке Си отечественных 32-разрядных микроконтроллеров семейства 1986BE9x разработки и производства компании «Миландр». Основное внимание уделено работе со встроенными в микроконтроллеры периферийными устройствами: линиями и портами ввода-вывода, аналого-цифровыми и цифро-аналоговыми преобразователями, аппаратными таймерами/счетчиками, часами реального времени.

Предполагается использование отладочной платы для микроконтроллера K1986BE92QI, системы программирования Keil µVision и операционной системы реального времени Keil RTX.

Содержатся задания для практической работы, методика их выполнения, примеры программ и контрольные вопросы для самопроверки.

Предназначено для студентов технических высших учебных заведений очной и заочной формы обучения, аспирантов, инженерно-технических работников.

Благодаров Андрей Витальевич
Владимиров Леонид Леонидович

Программирование микроконтроллеров
на основе отечественных микросхем семейства 1986BE9x
разработки и производства компании «Миландр»

Содержание

Введение

С 70-х годов разработчики систем управления стали использовать вычислительные системы на базе **микропроцессоров**, выпуск которых был освоен рядом производителей, таких как Intel, Texas Instruments, Motorola и др. Применение этой технологии разработки систем управления позволяло повысить скорость и эффективность проектирования новых систем на базе старых, снизить затраты на обнаружение и устранение неисправностей, а также удешевить производство.

Однако в силу своих архитектурных ограничений микропроцессоры не обладали возможностью непосредственно решать задачи управления, и для достижения поставленных целей разработчики вынуждены были снабжать их набором дополнительных устройств: памятью программ и данных, а также набором периферийных элементов: счетчиками, аналого-цифровыми и цифро-аналоговыми преобразователями, программируемыми контроллерами ввода-вывода и т.д.

Структура системы управления с применением описанных периферийных элементов употреблялась в разработках достаточно часто, в связи с чем возникла идея интеграции этих элементов на одном кристалле. Подобная идея применялась и ранее, что дало разработчикам возможность скомпоновать набор транзисторов в виде интегральной микросхемы.

Воплощение идеи произошло в 1976 году с выпуском фирмой Intel устройства под кодовым обозначением 8048, позднее получившего название **микроконтроллер** и ставшего основой систем управления, встраеваемых в робототехнические комплексы, бытовую электронику и др.

Под микроконтроллером здесь и далее подразумевается программируемое вычислительное устройство, обладающее набором периферийных устройств и применяемое для решения задач управления в технических системах.

Появившись на рынке, микроконтроллеры наращивали свою популярность и в настоящее время применяются чрезвычайно широко.

По области прменения, структурной организации, разрядности, набору периферийных устройств, системе команд и прочим признакам микроконтроллеры сгруппированы в **семейства**, число которых достаточной велико [1].

При освоении простейших микроконтроллеров для создания программного проекта зачастую достаточно изучить соответствующую документацию. Однако такой метод, по мнению автора, слабо применим при работе с 32-разрядными микроконтроллерами. Во-первых, он требует высоких временных затрат в виду большого объема информации. Во-вторых, для конфигурации периферии микроконтроллера необходимо вносить множество данных в различные регистры, не имея порой обратной связи. В таких случаях отсутствуют средства диагностики – нет возможности отследить место потери сигнала или некорректную работу подситемы, то есть приходится работать почти «вслепую».

Для изучения 32-разрядных микроконтроллеров отправной точкой должны служить работающие проекты, покрывающие некоторую ограниченную функциональность микроконтроллера, в комплексе с документацией и инструментами анализа. Изучение микроконтроллеров происходит путем внесения в программы проектов небольших изменений и постоянного контроля их работоспособности. Такой метод называется **обратной разработкой**, или, как принято его называть в зарубежной литературе, **reverse engineering.** Комлексные проекты при этом могут создаваться путем синтеза исходных кодов базовых проектов.

Данная книга посвящена отечественным микроконтроллерам семейства 1986ВЕ9х разработки и производства компании «Миландр». Книга представляет собой практикум по аппаратному программированию и содержит семь тем, охватывающих следующие основные аспекты в работе с микроконтроллерами:

- понятие отладочной платы для микроконтроллера, понятие среды программирования;
- использование портов ввода-вывода общего назначения;
- работа с аналого-цифровым преобразователем;
- работа с цифро-аналоговым преобразователем;
- реализация широтно-импульсной модуляции;
- использование аппаратных таймеров/счетчиков в режиме захвата;
- использование часов реального времени и батарейного домена.

Широко используется метод обратной разработки, а также механизм прямого доступа к памяти применительно к различным периферийным устройствам.

Вопросы, связанные с программированием различных интерфейсов, не рассматриваются: предполагается издание отдельной книги на эту тему.

К каждой работе подготовлены примеры проектов на языке Си, исходный код которых подробно прокомментирован. Из-за значительного объема разместить полные исходные коды проектов в книге не представляется возможным. Проекты могут быть бесплатно скачаны в официальной группе компании «Миландр» социальной сети «ВКонтакте» по ссылке [2].

Автор придерживается следующих основных принципов при программировании микроконтроллеров, что нашло отражение в данной книге:

1. Используется язык Си. Язык Си является практически безальтернативным вариантом при программировании современных микроконтроллеров.

2. Не используются возможности языка C++. Это обусловлено тем, что в программах на языке C++, построенных с применением объектно-ориентированного подхода, необходимо задействовать динамическое распределение памяти, что неизбежного приводит к снижению надежности программного обеспечения.

3. Не используется ассемблер. Это обусловлено тем, что при программировании мощных современных микроконтроллеров применение ассемблера стало неактуальным. Поэтому тратить время и силы читателя на изучение заведомо устаревших подходов нерационально.

4. Используется система программирования Keil µVision MDK-ARM. Данная система программирования выбрана исходя из ее популярности, развитости функционала, а также возможности бесплатно использовать ее в пробном режиме.

5. Используется стандартная периферийная библиотека. В учебных примерах работа с периферийными устройствами практически полностью осуществляется через стандартную периферийную библиотеку от компании «Миландр». Прямое обращение к регистрам устройств почти не используется. Это упрощает и ускоряет процесс знакомства с микроконтроллером.

6. Используется операционная система реального времени Keil RTX. RTX представляет собой систему с вытесняющей многозадачностью и развитыми средствами синхронизации задач. Применение такой системы позволяет наиболее полно и рационально задействовать возможности микроконтроллеров с архитектурой ARM32.

По мнению автора, такой подход позволяет быстро освоить основы программирования микроконтроллеров, научившись создавать надежный и красивый программный код.

Предполагается, что читатель хотя бы немного знаком с программированием на языке Си, причем необязательно применительно к микроконтроллерам.

Для работы требуется следующее основное оборудование:

1. Отладочный комплект для микроконтроллера К1986ВЕ92QI, выпускаемый компанией «Миландр».

2. Программатор-отладчик MT-Link, выпускаемый фирмой MT System (г. Санкт-Петербург). Подойдут и другие программаторы для микроконтроллеров с архитектурой ARM32, например: J-Link, ST-Link, Keil ULink2. Естественно, перед приобретением программатора нужно уточнить у производителя, совместим ли он с микроконтроллерами семейства 1986ВЕ9х.

3. Цифровой осциллограф-приставка USB-Oscill, выпускаемый в г. Одесса [3]. Можно использовать и другие цифровые или аналоговые осциллографы с полосой пропускания не менее 2 МГц. Осваивать программирование микроконтроллеров без осциллографа затруднительно.

4. Периферийный модуль, подключаемый к отладочной плате с помощью штыревых разъемов и соединительных проводов. Модуль включает в себя светодиоды, потенциометр и лампу накаливания. При его отсутствии отдельные области модуля вполне могут быть смонтированы самостоятельно.

О программном обеспечении, которое потребуется при работе с книгой, говорится в разделе «Установка программного обеспечения». Здесь лишь отметим, что все используемое программное обеспечение, за исключением операционной системы MS Windows, доступно бесплатно.

Установка программного обеспечения

Для работы с книгой потребуется следующее программное обеспечение, которое нужно установить на вашем компьютере:

1. Операционная система Microsoft Windows XP / 7 / 8 / 10.
2. Примеры проектов.
3. Среда программирования Keil µVision MDK-ARM версии 4.72 или выше.
4. Драйвер программатора-отладчика MT-Link.
5. Программа Windows Oscill для осциллографа-приставки.
6. Драйвер для осциллографа-приставки.

1. Операционная система.

Для корректной работы комплекса программного обеспечения необходима операционная система Windows XP / 7 / 8 / 10. Следует отметить, что на операционных системах Windows 7 / 8 / 10 процесс установки драйвера для осциллографа-приставки более трудоемкий, о чем рассказывается далее.

Использование операционной системы Linux, к сожалению, невозможно.

2. Примеры проектов.

Архив с примерами может быть свободно скачан из Интернета по ссылке [4]. Этот архив также содержит стандартную периферийную библиотеку для микроконтроллеров семейства 1986BE9x.

3. Среда программирования Keil µVision MDK-ARM.

Компания Keil предоставляет возможность использования бесплатной ознакомительной версии своего продукта с ограничением по размеру кода в 32 Кбайта флеш-памяти. Этого объема вполне достаточно для работы с простыми проектами.

Все примеры проектов, прилагаемые к книге, создавались в среде программирования Keil µVision MDK-ARM 4.72. На момент завершения книги поддержка и распространение версии 4.72 были прекращены, но вместо нее на официальном сайте компании Keil доступна новая версия продукта Keil µVision MDK-ARM 5.20 [5]. Все примеры программ, приводимые в книге, совместимы с этой версией. Для загрузки установочного файла потребуется формальная регистрация.

Установка программы очень проста и не вызывает затруднений. Достаточно запустить скачанный файл `MDK520.exe` и следовать указаниям инсталлятора. При выборе пути установки рекомендуется указать `C:\Keil` для синхронизации с книгой.

Далее необходимо установить дополнительный программный пакет, обеспечивающий совместимость среды с проектами, выполненными в ранних версиях Keil µVision MDK-ARM, поскольку предоставляемые примеры проектов разработаны с использованием версии 4.72. Для получения пакета следует зайти на страницу [6] и воспользоваться ссылкой Download Legacy Support for Cortex-M Devices. Для выполнения установки запустите скачанный файл `MDKCM520.exe` и следуйте указаниям инсталлятора.

4. Драйвер программатора-отладчика MT-Link.

Для корректной работы USB-программатора MT-Link необходимо установить соответствующий драйвер. Для этого **потребуются права администратора**. При их отсутствии рекомендуется обратиться к системному администратору.

Установка драйвера требует выполнения следующей последовательности действий:

1. Подключить USB-кабель программатора к свободному USB-порту компьютера.

2. Открыть *Диспетчер устройств* (*Компьютер – Свойства – Диспетчер устройств*).

3. Найти неопределенное устройство *J-Link*, кликнуть по нему правой кнопкой мыши и выбрать пункт *Обновить драйверы* (рисунок 0.1).

4. Выбрать пункт *Ручной поиск* и указать путь `C:\Keil`. При этом необходимо убедиться, что установлена галочка *Включая вложенные папки* (рисунок 0.2).

5. По завершении установки убедиться, что устройство *J-Link* определено и находится в группе *Контроллеры USB*.

Одним из показателей успешной установки драйвера является непрерывное свечение светодиода, видимого сквозь термоусадочное покрытие, на плате программатора.

В некоторых случаях установка драйвера происходит автоматически при подключении устройства к персональному компьютеру.

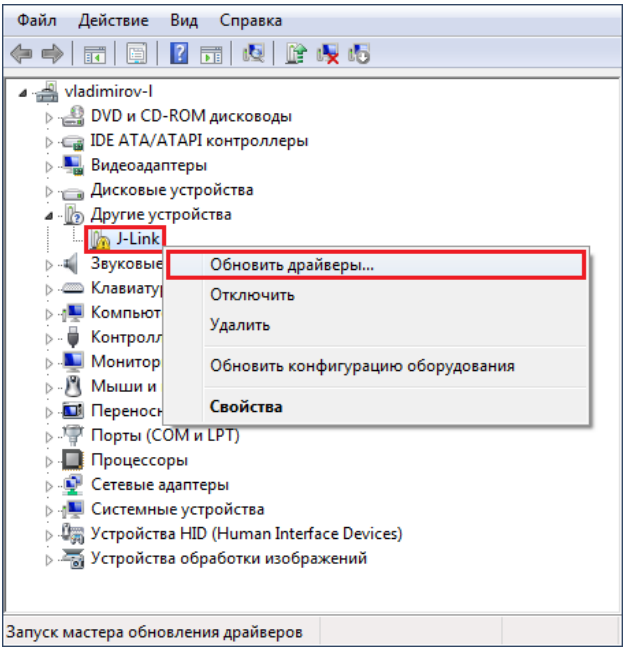


Рисунок 0.1 – Установка драйверов для программатора

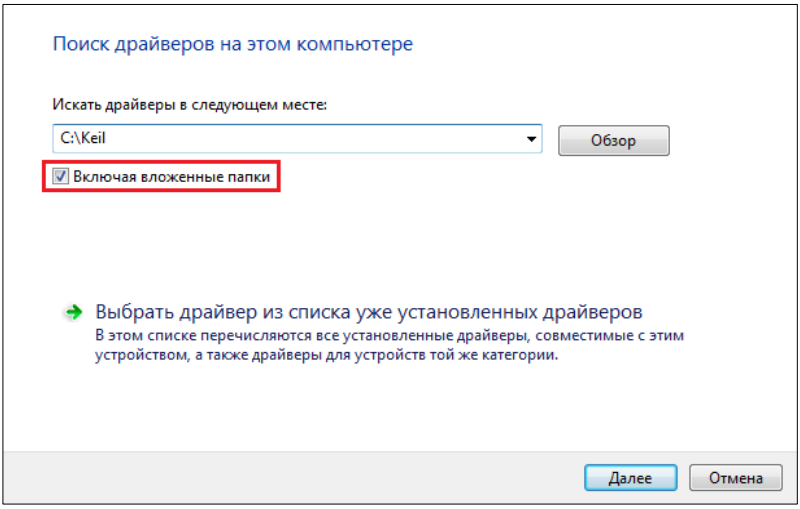


Рисунок 0.2 – Указание пути к драйверам для программатора

5. Программа Windows Oscill для осциллографа-приставки.

Программа необходима при использовании осциллографа-приставки для интерпретации и отображения результатов измерений на дисплей компьютера. Осциллограф потребуется для выполнения работ с цифро-аналоговым преобразователем, широтно-импульсной модуляции и аппаратными таймерами/счетчиками.

Для установки программы скачайте архив, расположенный по адресу [7] и распакуйте его, например, в корень диска `C:\` так, чтобы программа располагалась по адресу `C:\Oscill`.

Запустите самораспаковывающийся архив `Msvbvm50.exe`, расположенный в папке с программой. Он добавит в систему Windows необходимую для работы программы библиотеку.

6. Драйвер для осциллографа-приставки.

Для правильной идентификации осциллографа системой Windows необходима установка соответствующего драйвера. Трудность заключается в том, что нужный драйвер не имеет цифровой подписи. Операционные системы Windows 7 / 8 / 10 по-умолчанию требуют обязательного наличия цифровой подписи для выполнения установки (**система Windows XP лишена этой особенности**). Поэтому для нормального функционирования осциллографа проверку цифровой подписи драйверов необходимо отключить.

Для этого нужно **запустить от имени администратора** файл `cmd.exe`, расположенный по адресу `C:\Windows\System32` и набрать в командной строке следующий текст:

```
bcdedit.exe /set loadoptions ddisable_integrity_checks
bcdedit.exe /set nointegritychecks on
```

В итоге командная строка должна выглядеть, как показано на рисунке 0.3.

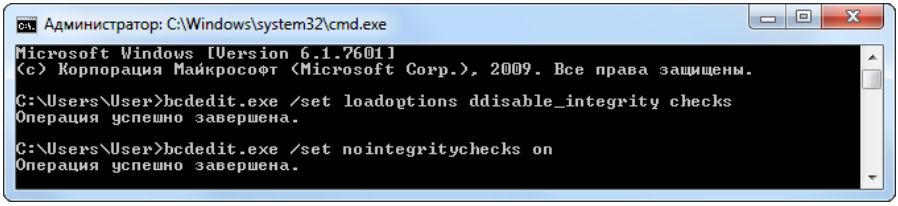


Рисунок 0.3 – Вид командной строки

Далее необходимо выполнить загрузку операционной системы без обязательной проверки цифровой подписи. Делается это по-разному в зависимости от операционной системы.

Для Windows 7 необходимо выполнить перезагрузку компьютера, и при загрузке системы периодически нажимать клавишу F8 до тех пор, пока не появится меню выбора режима загрузки. В меню необходимо выбрать пункт *Отключение обязательной проверки цифровой подписи драйверов*.

Для Windows 8 следует нажать сочетание клавиш *Win + I*, и в появившемся меню выбрать пункт *Параметры*. Далее нужно зажать клавишу *Shift* и последовательно выполнить команды *Выключение – Перезагрузка – Диагностика – Дополнительные параметры – Параметры загрузки – Перезагрузить*. Во время перезагрузки системы появится окно с выбором параметров загрузки. Необходимо выбрать параметр *Отключить обязательную проверку подписи драйверов*, нажав клавишу *F7*.

Наконец следует непосредственно установить драйвер осциллографа. Установка драйвера для осциллографа очень похожа на установку драйвера для программатора и требует выполнения следующей последовательности действий:

1. Подключить USB-кабель осциллографа к свободному USB-порту компьютера.

2. Открыть *Диспетчер устройств* (*Компьютер – Свойства – Диспетчер устройств*).

3. Найти неопределенное устройство *USB Oscill interface*, кликнуть по нему правой кнопкой мыши и выбрать пункт *Обновить драйверы*.

4. Выбрать пункт *Ручной поиск* и указать путь `C:\Oscill\drivers\usb33`. При этом необходимо убедиться, что установлена галочка *Включая вложенные папки*.

5. По завершении установки убедиться, что устройство *Oscill USB Device* определено и находится в группе *Контроллеры USB*.

Глава 1

Отладочная плата для микроконтроллера K1986BE92QI и среда программирования Keil µVision

Цель работы:

- знакомство с демонстрационно-отладочной платой для микроконтроллера K1986BE92QI;
- получение навыков работы в среде Keil µVision;
- получение представления о структуре проекта на языке Си.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT- Link;
- примеры кода программ.

1.1. Подготовка к работе

Перед непосредственной работой с проектами создайте на локальном диске папку, указав в качестве ее имени ваши инициалы, и скопируйте в нее папку `Samples` со всем ее содержимым. Дальнейшие действия выполняйте с копиями проектов из новой папки. Это позволит восстановить исходный проект в случае необходимости.

1.2. Описание проектов

В примере `Lab1_1` выполняется мигание двумя светодиодами, находящимися на отладочной плате.

В примере `Lab1_2` выполняется вывод бегущей строки на жидкокристаллический индикатор (ЖКИ).

1.3. Сведения о микроконтроллере K1986BE92QI

В рамках данного практикума будет использован микроконтроллер K1986BE92QI отечественной компании «ПКК Миландр» (www.milandr.ru). Микросхема K1986BE92QI является мощным 32-разрядным микроконтроллером с процессорным ядром семейства ARM32 Cortex-M3 и развитым набором периферийных устройств. Ниже приведены основные характеристики микроконтроллера:

- разрядность шины данных – 32 бита;
- процессорное ядро – ARM32 Cortex M3;
- объем флеш-памяти программ – 128 Кбайт;
- объем статической оперативной памяти данных – 32 Кбайт;
- тактовая частота процессорного ядра – до 80 МГц;
- ток потребления в активном режиме при тактовой частоте ядра 80 МГц - 120 мА;
- напряжение питания – 2,2…3,6 В;
- интерфейсы для программирования и отладки – JTAG и SWD.

В состав микроконтроллера входят следующие периферийные устройства:

- 43 линии ввода-вывода общего назначения, объединенные в 6 портов;
- 2 аналого-цифровых преобразователя (12 бит, 8 каналов);
- 1 цифро-аналоговый преобразователь (12 бит);
- контроллер прямого доступа к памяти;
- 3 аппаратных таймера/счетчика общего назначения (каждый по 16 бит);
- часы реального времени;
- 1 интерфейс USB 2.0;
- 2 интерфейса USART;
- 2 интерфейса SPI;
- 1 интерфейс I2C;
- 2 интерфейса CAN.

Познакомиться подробнее с описанием микроконтроллера K1986BE92QI можно по фирменной документации, приведенной в файле `Docs\1986BE9x.pdf` или на сайте компании «ПКК Миландр».

1.4. Сведения об отладочной плате

Отладочная плата для микроконтроллера K1986BE92QI, выпускаемая компанией «Миландр», предназначена для ознакомления с возможностями микроконтроллера и отладки программного обеспечения для него. Внешний вид отладочной платы показан на рисунке 1.1.

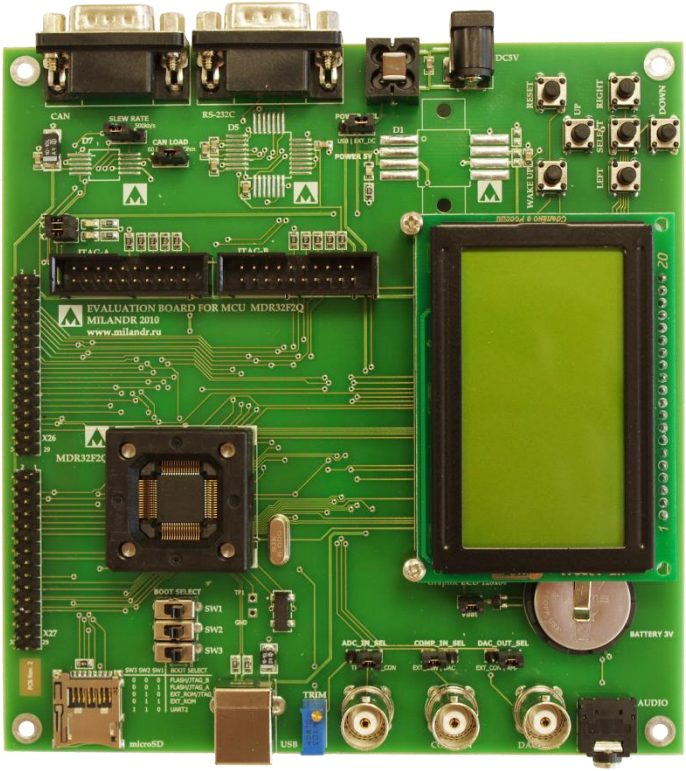


Рисунок 1.1 – Внешний вид отладочной платы для микроконтроллера K1986BE92QI

Питание платы, как правило, осуществляется от внешнего блока питания, входящего в состав отладочного комплекта. Для этого на плате предусмотрен соответствующий разъем. Блок питания подключается к сети переменного тока ~220 В, 50 Гц и выдает постоянное напряжение +5 В при токе до 0,5 А.

Также возможно питание платы от USB-интерфейса. В этом случае плата должна быть соединена с персональным компьютером с помощью USB-кабеля.

На рисунке 1.2 показано размещение основных элементов на плате 1986EvBrd_64, а в таблице 1.1 приведено их описание.

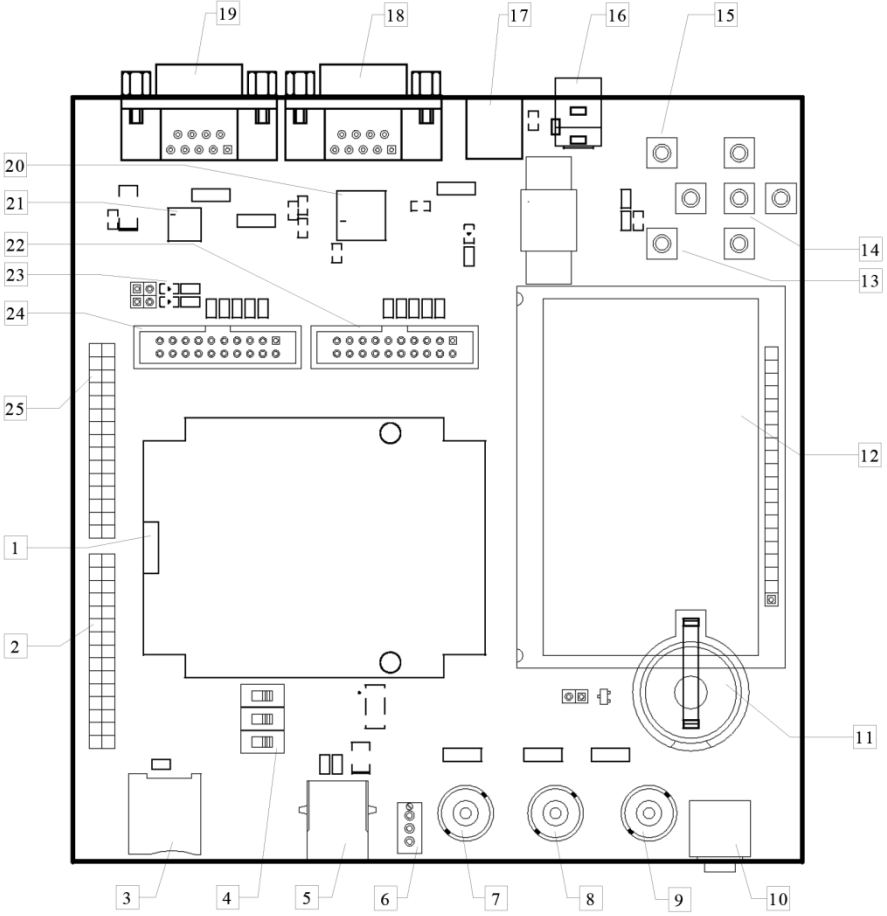


Рисунок 1.2 – Размещение основных элементов на отладочной плате

Таблица 1.1 – Описание элементов на плате 1986EvBrd_64

№	Наименование элемента
1	Контактирующее устройство для микроконтроллера K1986BE92QI
2	Разъем X27 для портов A, E, F микроконтроллера
3	Разъем карты памяти microSD
4	Переключатели выбора режима загрузки
5	Разъем USB-B
6	Подстроечный резистор на 7-м канале АЦП
7	Разъем BNC внешнего сигнала на 7-м канале АЦП
8	Разъем BNC внешнего сигнала на 1-м входе компаратора
9	Разъем BNC на выходе ЦАП1
10	Разъем TRS 3,5 мм на выходе усилителя звуковой частоты
11	Батарея 3,0 В
12	Жидкокристаллический индикатор с разрешением 128x64
13	Кнопка WAKEUP
14	Кнопки UP, DOWN, LEFT, RIGHT, SELECT
15	Кнопка RESET
16	Разъем питания 5 В
17	Фильтр питания
18	Разъем RS-232
19	Разъем CAN
20	Приемо-передатчик интерфейса RS-232 на микросхеме 5559ИН4
21	Приемо-передатчик интерфейса CAN на микросхеме 5559ИН14
22	Разъем программирования и отладки JTAG-B
23	Набор из двух красных светодиодов на порте C
24	Разъем программирования и отладки JTAG-A
25	Разъем X26 для портов B, C, D микроконтроллера

Для загрузки программ, написанных с помощью персонального компьютера, во флеш-память микроконтроллера, а также для их отладки, необходим программатор-отладчик. В работах будет использован программатор-отладчик (далее – программатор) MT-Link отечественной фирмы MT-SYSTEM [8], показанный на рисунке 1.3. Он является аналогом известного программатора J-Link фирмы IAR-Systems. Программатор подключается к компьютеру с помощью USB-кабеля и использует интерфейсы для внутрисхемной отладки SWD – Serial Wire Debug или JTAG. На плате предусмотрено два разъема для подключения программатора (JTAG-A и JTAG-B).

Рисунок 1.3 – Программатор-отладчик MT- Link

Почти все линии ввода-вывода микроконтроллера заведены на штырьковые разъемы X26 и X27, что позволяет подключить к ним необходимые внешние устройства. Назначение каждого контакта приведено в таблице 1.2. Также на эти разъемы заведены цепи земли (GND) и питания (+5 В, +3,3 В), от которых можно запитать подключаемые к плате внешние схемы.

Таблица 1.2 – Назначение контактов разъемов X26 и X27

Контакт	**Назначение**	
	X26	**X27**
1, 2	GND	GND
3, 4	+3,3 V	+3,3 V
5	PD0	PA6
6	PD1	PA7
7	PD2	PA4
8	PD3	PA5
9	PD4	PA2
10	PD5	PA3
11	PD6	PA0
12	–	PA1
13	PB0	–
14	PB1	–
15	PB2	PE1
16	PB3	PE3
17	PB4	–
18	PB5	–
19	PB6	PF0
20	PB7	PF1
21	PB8	PF2
22	PB9	PF3
23	PB10	PF4
24	PC0	PF5
25	PC1	PF6
26	PC2	–
27, 28	+5 V	+5 V
29, 30	GND	GND

Для отображения буквенно-цифровой и графической информации на плате предусмотрен монохромный жидкокристаллический индикатор размером 128x64 пикселя. При отладке приложений удобно пользоваться двумя светодиодами красного цвета, подключенными к выводам микроконтроллера.

Для ввода информации можно использовать пять механических кнопок общего назначения: «LEFT», «RIGHT», «UP», «DOWN» и «SELECT». Для сброса и перезапуска микроконтроллера предназначена кнопка «RESET».

На плате также предусмотрен целый ряд иных компонентов, которые будут постепенно изучаться в последующих лабораторных работах. Более подробную информацию о плате 1986EvBrd_64 можно получить из фирменного описания, которое доступно по ссылке [9]. Принципиальная схема отладочной платы приведена в файле [10].

1.5. Подготовка отладочной платы к работе

Для подготовки отладочной платы к работе следует выполнить следующие действия:

1. Откройте коробку с отладочным комплектом, извлеките из нее плату и положите на стол.
2. Отсоедините от платы дополнительные провода и детали, если таковые подключены к штыревым разъемам X26 и X27.
3. Возьмите из коробки программатор MT-Link и USB-кабель, и соедините их между собой.
4. Подключите шлейф программатора к разъему JTAG-B, расположенному на плате. Обратите внимание на специальную направляющую, исключающую неправильное подключение.
5. Убедитесь, что каждый из трех переключателей выбора режима загрузки (позиция 4 на рисунке 1.2) установлен в положение «0», что соответствует использованию интерфейса JTAG-B при отладке.
6. Включите компьютер, и после загрузки системы подключите к свободному USB-порту компьютера USB-кабель программатора. На программаторе должен загореться светодиод, который будет видно сквозь термоусадочную трубку, покрывающую программатор. Если светодиод все время мигает, то это означает, что нет связи между компьютером и программатором. Чаще всего такая проблема возникает ввиду отсутствия необходимого драйвера. Инструкция по его установке приведена в соответствующем разделе.

7. Убедитесь, что перемычка «POWER_SEL» установлена в положение «EXT_DC» (внешний источник питания). При необходимости переставьте ее в это положение. При другом положении перемычки будет использовано питание со стороны USB-интерфейса.

8. Включите блок питания в сетевую розетку и подключите шнур питания к соответствующему разъему на плате (позиция 16 на рисунке 1.2). На плате должен загореться красный светодиод «POWER 5V». Если в микроконтроллере уже содержится программа, то она начнет выполняться. Возможно, это будет сопровождаться миганием красных светодиодов и выводом информации на ЖКИ.

1.6. Среда программирования Keil µVision

Интегрированная среда программирования Keil µVision MDK-ARM предназначена для написания и отладки программ для микроконтроллеров семейства ARM32 с помощью языков Си, C++ и ассемблера. В рамках нашей работы будем использовать лишь язык Си.

В состав среды входят все необходимые для этого средства: специализированный текстовый редактор с семантической (смысловой) подсветкой кода, компилятор, ассемблер, компоновщик, отладчик и т.д. Среда программирования поддерживает практически все выпускаемые в мире микроконтроллеры с архитектурой ARM32. Keil µVision посредством драйверов может работать с различными внутрисхемными программаторами-отладчиками, в том числе и с MT- Link.

Если на вашем компьютере не установлена среда программирования Keil µVision, то следует обратиться к разделу «Установка программного обеспечения».

Запустите среду Keil µVision с помощью ярлыка на рабочем столе, или, в случае его отсутсвия, запустите файл `UV4.exe`, расположенный по адресу `C:\Keil\UV4`.

Внешним видом среда Keil µVision напоминает среду Microsoft Visual Studio, поэтому программистам, знакомым с Visual Studio, нетрудно будет освоиться и со средой Keil. К сожалению, русифицированной версии Keil µVision пока не появилось. Весь интерфейс организован на английском языке.

Для знакомства со средой откроем готовый учебный проект, выполнив пункт главного меню *Project – Open Project*. В появившемся диалоговом окне выберем и откроем файл проекта, расположенный по адресу `Samples\Project\Lab1_1\MDK-ARM\Project.uvproj`.

На рисунке 1.6 показан вид главного окна с открытым проектом. В левой части располагается окно *Project* со структурой проекта. В правой части – вкладки с открытыми исходными текстами программ. В нижней части размещается окно *Build Output* для сообщений о ходе построения проекта. На панель инструментов выведены кнопки для доступа к наиболее часто используемым операциям.

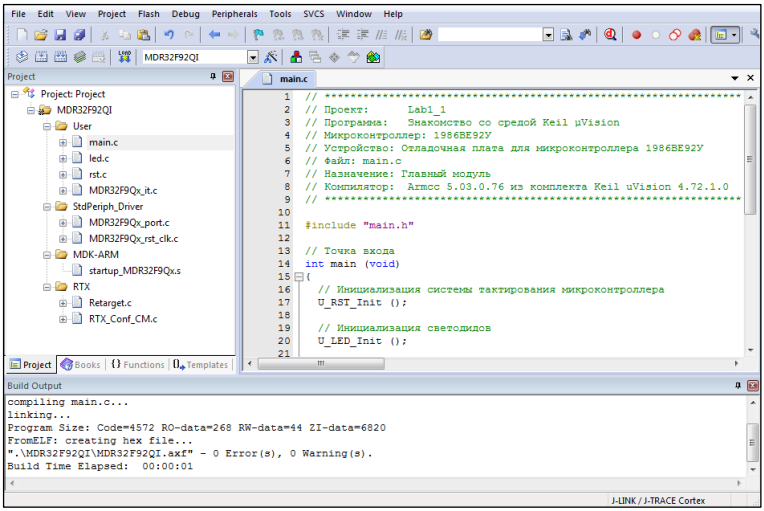


Рисунок 1.6 – Главное окно Keil µVision

Конечно, программист при желании может переконфигурировать внешний вид главного окна, но **на учебных компьютерах этого, пожалуйста, не делайте.**

Открытый проект написан на языке Си и решает следующую задачу: мигание двумя светодиодами. Каждый светодиод мигает со своей частотой.

Обратите внимание на поясняющий текст, выделенный зеленым цветом. Такой текст называют комментарием. Он не учитывается при компиляции, поэтому нет причин жертвовать подробностью таких пояснений

ради оптимизации кода. Комментарии бывают многострочными (начинаются с символов /* и заканчиваются символами */) и однострочными (начинаются с символов //).

```
/* Многострочный комментарий нередко размещают в начале файла,
где он содержит имя автора и описание программы */

#include "MDR32Fx.h" // Подключаем заголовочный файл MDR32Fx.h
```

Построим проект, нажав клавишу *F7*. Под построением понимают компиляцию всех модулей, входящих в состав проекта, их ассемблирование и компоновку. То же действие доступно и через пункт главного меню *Project – Build target* или через кнопку  на панели инструментов. Заметим, что иногда полезно бывает полностью перестроить проект, выбрав пункт меню *Project – Rebuild all target files* или нажав кнопку  (горячая клавиша не предусмотрена). Разница между построением и перестроением проекта заключается в том, что в процессе перестроения заново компилируются все модули, а при построении – только те, которые подверглись изменению после последней компиляции. Перестроение занимает больше времени, но работает надежней.

Результатом построения проекта является так называемый HEX-файл, предназначенный для загрузки в память микроконтроллера. Формат HEX, предложенный в свое время фирмой Intel, предназначен для представления произвольных двоичных данных в текстовом виде. Если открыть такой файл обычным текстовым редактором, то он будет состоять из строк шестнадцатеричных цифр, поэтому его и называют «HEX» – от английского слова «hexadecimal» – «шестнадцатеричная система счисления».

Результаты построения отображаются в нижнем окне *Build Output*. Далее приведен в сокращенном виде пример сообщений, полученных при полном перестроении проекта.

```
Rebuild target 'MDR32F92QI'
compiling main.c...
compiling led.c...
compiling rst.c...
compiling MDR32F9Qx_it.c...
compiling MDR32F9Qx_port.c...
compiling MDR32F9Qx_rst_clk.c...
...
assembling startup_MDR32F9Qx.s...
compiling Retarget.c...
compiling RTX_Conf_CM.c...
linking...
Program Size: Code=4524 RO-data=268 RW-data=44 ZI-data=6820
".\MDR32F92QI\MDR32F92QI.axf" - 0 Errors, 0 Warning(s).
```

Как видно из примера, сначала были откомпилированы все модули на языке Си (файлы с расширением `*.c`) и ассемблирован вспомогательный модуль `startup_MDR32F9Qx.s`, написанный на языке ассемблер. Затем все модули были скомпонованы. Процесс построения прошел без ошибок и замечаний: `0 Errors, 0 Warning(s)`. Если же в процессе компиляции или компоновки возникнут ошибки, что часто бывает в процессе написания и отладки программ, то соответствующие сообщения появятся в этом окне.

В строке *Program Size* приводятся важнейшие характеристики полученной программы – занимаемая ей память и ее объем:

- `Code=4572` – программный код занимает 4572 байта флеш-памяти;
- `RO-data=268` – постоянные данные (различные константы) занимают 268 байт флеш-памяти;
- `RW-data=44` – оперативные данные (переменные), которые инициализируются ненулевыми значениями, занимают 44 байта ОЗУ. Для хранения инициализационных значений также приходится выделить 44 байта во флеш-памяти;
- `ZI-data=6820` – оперативные данные (переменные), которые инициализируются нулями, занимают 6820 байта ОЗУ.

Таким образом, общий объем требуемой флеш-памяти составляет:
`Code + RO + RW = 4572 + 268 + 44 = 4884` (байта).
Общий объем требуемой оперативной памяти составляет:
`ZI + RW = 6820 + 44 = 6864` (байта).

Создавая проект надо всегда помнить, что объем памяти микроконтроллера ограничен, и нужно укладываться в установленные рамки. Напомним, что для нашего микроконтроллера K1986BE92QI объем флеш-памяти программ составляет 128 Кбайт, а оперативной памяти данных – 32 Кбайт. Кроме того, в ознакомительной версии Keil µVision MDK-ARM, которой вы сейчас пользуетесь, установлено **ограничение в 32 Кбайта** флеш-памяти. Вплотную к этим показателям приближаться не следует. Лучше всегда иметь запас хотя бы в 10%.

Если памяти не хватает, то можно порекомендовать такие действия:

- оптимизировать программный код по объему, используя соответствующую директиву компилятора;
- использовать микроконтроллер того же семейства, но с большим объемом флеш-памяти программ или ОЗУ;
- использовать менее ресурсоемкие алгоритмы.

Результатом построения проекта являются файлы `Lab1_1.axf` и `Lab1_1.hex`. Файл `Lab1_1.axf` содержит в себе исполняемый код и необходимую отладочную информацию. Он применяется для отладки программы в среде Keil. Файл `Lab1_1.hex`, как уже говорилось, содержит готовую «прошивку» для микроконтроллера в формате HEX. Стоит отметить, что имея готовый HEX-файл, можно загружать в микроконтроллер программу и без среды Keil, используя другие программные средства.

1.7. Структура проекта в Keil µVision

Проект программы для микроконтроллера в среде Keil µVision представляет собой достаточно сложную совокупность файлов, каталогов и настроек. Поэтому при начальном знакомстве со средой Keil µVision рекомендуется не создавать новый проект, проводя многочисленные настройки, а воспользоваться уже созданным проектом, внеся в него необходимые изменения. Впрочем, такой подход широко используется и при разработке серьезных приложений.

1.7.1. Модули проекта

Во-первых, проект состоит из модулей, написанных на языках Си, C++ или ассемблер. Напомним, что модуль на языке Си, как правило, состоит из

двух файлов: собственно модуль – файл с расширением *.c и заголовочный файл с расширением *.h. Имена же обоих файлов одинаковые. В модуле содержатся исходные коды функций и объявления глобальных переменных, а в заголовке – прототипы (предварительные описания) функций и глобальных переменных. Подключив заголовочный файл директивой #include к другому модулю, можно задействовать в одном модуле функции и переменные другого модуля. В отдельном модуле обычно располагают функции, родственные по смыслу, например, относящиеся к работе с одним устройством или к одной задаче.

Заголовочный файл всегда начинается с директивы условной компиляции #ifndef с проверкой существования определенной константы. Если константа еще не объявлена, это значит, что компилятор впервые увидел этот заголовок и будет произведена его дальнейшая компиляция. Если константа уже объявлена, то дальнейшая компиляция заголовка не происходит. Это предотвращает многократное определение одних и тех же функций и переменных.

Следующим шагом в заголовке директивой #define объявляется такая константа. Заканчивается заголовок директивой #endif (конец условной компиляции).

Рассмотрим, к примеру, модуль led.c, отвечающий за работу со светодиодами. Вот его заголовок led.h в сокращенном виде:

```
#ifndef __U_LED
#define __U_LED

#include "common.h"
...
// Переключить указанные светодиоды
void U_LED_Toggle (uint32_t Pins);
...
#endif
```

В качестве проверяемой константы используется имя __U_LED. Желательно давать таким константам имена по определенному правилу: __U_<имя модуля>. Это позволит избежать случайного совпадения имени константы с каким-нибудь иным идентификатором. Буква «U» означает здесь модуль (unit).

В нашем заголовке производится подключение еще одного заголовочного файла (common.h), нужного для работы модуля.

Далее приведен фрагмент самого модуля led.c:

```
#include "led.h"
...
// Переключить указанные светодиоды
void U_LED_Toggle (uint32_t Pins)
{
   uint32_t data = PORT_ReadInputData (U_LED_PORT);
   PORT_Write (U_LED_PORT, data ^= Pins);
}
```

Первым делом подключается соответствующий заголовочный файл, а затем идут тексты всех функций, в том числе обязательно тех, которые были ранее декларированы в заголовке.

Главным модулем проекта всегда является модуль main.c. В нем обязательно объявляется функция main(), с вызова которой начинается выполнение программы. Когда микроконтроллер включается, управление автоматически передается в функцию main(). При работе с микроконтроллером функция main() никогда не должна завершаться, т.е. должна в том или ином виде содержать бесконечный цикл. Перед таким циклом производится инициализация требуемых в проекте периферийных устройств микроконтроллера.

В простейшем случае функция main() выглядит так:

```
{
   // Инициализация устройства 1
   Device1_Init ();
   // Инициализация устройства 2
   Device2_Init ();
   ...
   // Инициализация устройства N
   DeviceN_Init ();

   while(1)
   {
      // Вызов задачи 1
      Task1 ();
      // Вызов задачи 2
      Task2 ();
      ...
```

```
        // Вызов задачи M
        TaskM (); 
    }
return 0;
}
```

Функции `Device1_Init`, `Task1` и другие определяются в отдельных модулях. Оператор `return` здесь поставлен для строгости, на самом деле выполнение программы до него никогда не дойдет.

Описанию функции `main`, как правило, предшествуют директивы подключения заголовков модулей, принадлежащих проекту, а также необходимых проекту заголовков библиотечных модулей. Чтобы можно было нормально работать возможностями микроконтроллера, обязательно подключается заголовок `common.h`. В нем, в свою очередь, содержатся директивы подключения заголовков всех модулей периферийной библиотеки, а также системных заголовков `MDR32Fx.h` и `RTL.h`:

```
// Работа с RTX
#include <RTL.h>

// Библиотеки для работы с периферией микроконтроллера 1986BE9x
#include "MDR32Fx.h"
#include "MDR32F9Qx_config.h"
#include "MDR32F9Qx_rst_clk.h"
#include "MDR32F9Qx_port.h"
#include "MDR32F9Qx_dac.h"
#include "MDR32F9Qx_bkp.h"
#include "MDR32F9Qx_ssp.h"
#include "MDR32F9Qx_uart.h"
#include "MDR32F9Qx_dma.h"
#include "MDR32F9Qx_adc.h"
#include "MDR32F9Qx_wwdg.h"
#include "MDR32F9Qx_adc.h"
#include "MDR32F9Qx_timer.h"
```

В заголовке `MDR32Fx.h` определены имена всех регистров, портов, устройств и т.д.

В нашем проекте задействована многозадачная операционная система реального времени (ОСРВ) RTX, интегрированная в среду Keil. ОСРВ выполняет важное дело – реализует **вытесняющую многозадачность**, сильно облегчая труд программиста. Функции, доступные в RTX описаны в заголовке `RTL.h`.

В связи с использованием RTX, вместо явного бесконечного цикла в функции main присутствует вызов функции-планировщика задач os_sys_init (Main_Task_Init), которая никогда не заканчивается. В качестве параметра передается указатель на функцию Main_Task_Init, определенную в модуле main.c. В эту функцию будет автоматически передано управление в момент запуска RTX.

Функция Main_Task_Init производит создание задач, выполняемых под управлением операционной системы. В нашем случае их две: «моргать светодиодом 0» (U_LED_Task0_Function) и «моргать светодиодом 1» (U_LED_Task1_Function). Задачи создаются путем вызова функции os_tsk_create, входящей в состав RTX. В качестве одного из параметров указывают имя функции, в которой реализована задача. По завершении функции Main_Task_Init обе задачи будут выполняться независимо друг от друга, а ОСРВ будет автоматически делить между ними процессорное время. Подробнее о различных возможностях RTX можно узнать из [11].

Далее приведен пример работы с RTX. Обратите внимание на специальный модификатор __task в заголовке функции. Так должны оформляться все задачи в RTX.

Задача, как правило, тоже содержит бесконечный цикл, внутри которого и выполняется.

```
#include "main.h"
// Точка входа
int main (void)

{
   // Инициализация системы тактирования микроконтроллера
   U_RST_Init ();

   // Инициализация светодиодов
   U_LED_Init ();

   // Инициализация задач RTX
   os_sys_init (Main_Task_Init);

   return 0;
}
```

```
// Инициализация задач RTX
__task void Main_Task_Init (void)

{
   // Повысить приоритет текущей задачи
   os_tsk_prio_self (100);
   // Создать задачу "Моргать светодиодом 0"
   os_tsk_create (U_LED_Task0_Function, 20);
   // Создать задачу "Моргать светодиодом 1"
   os_tsk_create (U_LED_Task1_Function, 20);
   // Удалить текущую задачу
   os_tsk_delete_self ();
}
```

Для удобства программиста в среде Keil модули объединяют в группы по назначению. Количество и названия групп можно определять по своему усмотрению. На рисунке 1.7 показана развернутая структура проекта с указанием всех модулей. Это изображение доступно в окне *Project*.

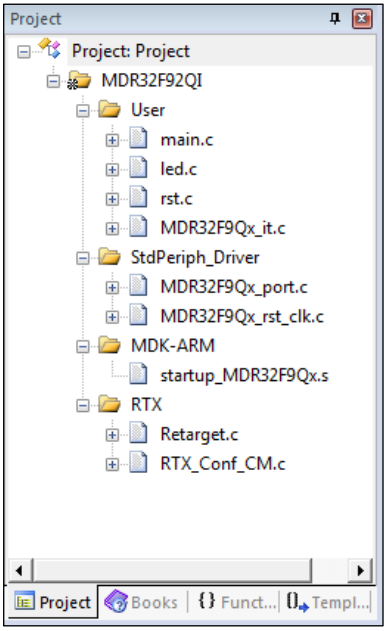


Рисунок 1.7 – Структура модулей проекта

Модули, созданные самим автором проекта, размещены в группе User. Сюда же добавлен файл MDR32F9Qx_it.c, в котором размещают обработчики аппаратных прерываний микроконтроллера.

В группе StdPeriph_Driver указаны библиотечные модули из стандартной библиотеки периферийных устройств, предоставляемой фирмой «Миландр». Эта библиотека содержит множество функций для работы с различными устройствами, входящими в состав микроконтроллера. Ее использование радикально упрощает написание программ.

Для добавления новой группы достаточно щелкнуть правой кнопкой мыши по корневому элементу дерева групп и выполнить пункт контекстного меню *Add Group*. Для добавления в группу существующего модуля, из контекстного меню для соответствующей группы выполняют действие *Add Existing Files to Group*. Если надо создать и добавить новый модуль, то аналогично выполняют действие *Add New Item to Group.* В открывшемся окне выбирают тип файла, указывают его имя и размещение. Для нового модуля автоматически откроется пустая вкладка.

Чтобы исключить модуль из проекта, надо выбрать его по дереву и выполнить пункт контекстного меню *Remove File*. Заметим, что это действие не приводит к удалению файла: он будет лишь исключен из проекта, оставшись на диске.

1.7.2. Файлы проекта

Во-вторых, проект состоит из совокупности различных файлов, размещенных в определенной структуре каталогов на диске. Типичная структура каталогов приведена на рисунке 1.8. Для знакомства с ней откройте *Проводник* и зайдите в каталог Samples, где хранятся учебные примеры программ для микроконтроллера. В каталоге Samples (в других проектах он, естественно, может именоваться по-другому) размещены подкаталоги Libraries (библиотеки) и Project (проект).

В подкаталоге Libraries лежат библиотеки, необходимые для создания проекта. Это – уже упомянутая стандартная библиотека периферийных устройств от фирмы «Миландр» для микроконтроллеров семейства 1986ВЕ9х (подкаталог MDR32F9Qx_StdPeriph_Driver) и CMSIS – библиотека для поддержки процессорного ядра всех микроконтроллеров,

совместимых с ARM32 Cortex. Основное в обеих библиотеках – модули на языке Си с соответствующим набором функций, т.е. файлы `*.c` и `*.h`.

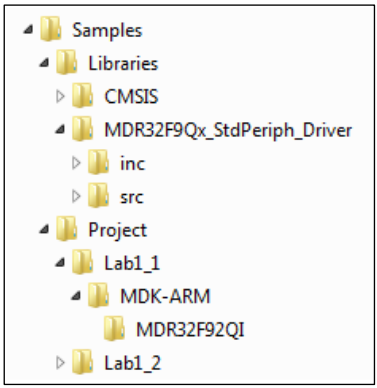


Рисунок 1.8 – Структура каталогов проекта

В каталоге `Project` лежат подкаталоги отдельных проектов, коих может быть произвольное количество. Как видно из рисунка 1.8, здесь их два: `Lab1_1` и `Lab1_2`.

Таким образом, все проекты используют одни и те же библиотеки из каталога `Libraries`, расположенного чуть выше. Это удобно с точки зрения экономии места на диске и простоты обновления библиотек.

Непосредственно в подкаталоге проекта `Lab1_1` размещены все файлы модулей `*.c` и заголовков `*.h`, которые созданы программистом в рамках проекта (из группы `User`). По сути, это – все исходники проекта. Сюда же помещают ранее упомянутые файлы из группы `RTX`: `RTX_Conf_CM.c` и `Retarget.c`. Из подкаталогов, представляющих для нас интерес, здесь есть подкаталог `MDK-ARM`, содержащий вспомогательные файлы для среды Keil. Для других сред программирования создаются отдельные подкаталоги. Таким образом пытаются достигнуть переносимости проекта между разными средами, что, впрочем, на практике достижимо далеко не всегда.

В подкаталоге `MDK-ARM` интересен файл проекта `Project.uvproj`. Он содержит все основные настройки проекта. Если этот файл попытаться открыть в *Проводнике*, то автоматически запустится среда Keil и загрузится соответствующий проект.

В подкаталоге `MDR32F92QI` (имя дано по названию отладочной платы) средой Keil создаются разнообразные вспомогательные файлы в процессе компиляции, ассемблирования и компоновки. Отметим лишь важность файлов с расширениями `*.axf`, `*.hex`, `*.map` и `*.sct`:

- `*.axf` – файл необходим для внутрисхемной отладки проекта;
- `*.hex` – файл прошивки микроконтроллера, т.е. то, ради чего все это и затевается;
- `*.sct` – текстовый файл, содержащий описание используемых проектом секций памяти;
- `*.map` – файл содержит полное описание распределения памяти для объектов программы (переменных, констант, функций); анализируя этот файл, можно определить количество памяти, задействованной в проекте, и цель ее использования.

Все, что содержится в подкаталоге `MDR32F92QI`, можно смело удалять: при построении проекта все будет возвращено. Однако не следует удалять файл `*.sct`, если вы правили его самостоятельно.

1.7.3. Основные настройки проекта

В-третьих, каждый проект имеет свои настройки. Рассмотрим основные.

Вернемся в среду Keil и зайдем в настройки проекта, который, естественно, должен быть открыт, выполнив пункт меню *Project – Options for Target* или нажав кнопку  на панели инструментов.

Перейдем на вкладку *Device* (устройство), как показано на рисунке 1.9. Здесь при помощи дерева выбирается тип микроконтроллера. В нашем случае – *MDR32F9Q2I* (это название микроконтроллера K1986BE92QI, используемое для международного рынка).

Далее перейдем на вкладку *Target* (цель проекта), показанную на рисунке 1.10. На этой вкладке задается тактовая частота микроконтроллера, при необходимости выбирается операционная система, а также указываются используемые в микроконтроллере диапазоны памяти.

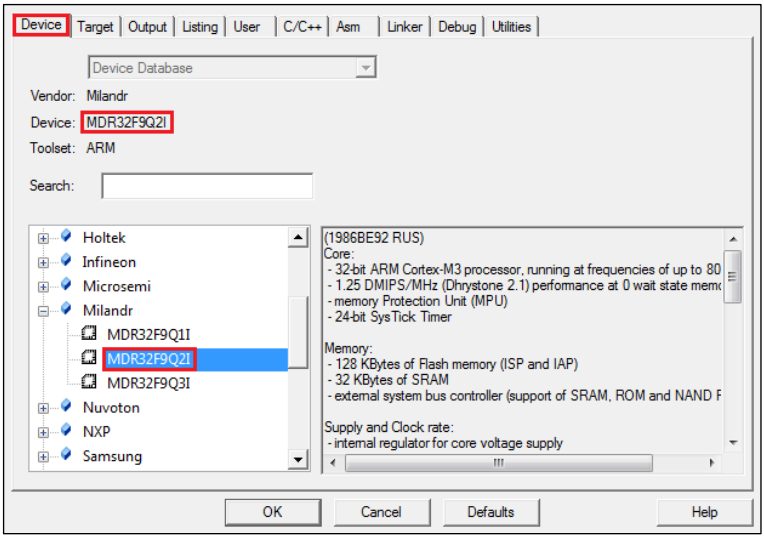


Рисунок 1.9 – Выбор типа микроконтроллера

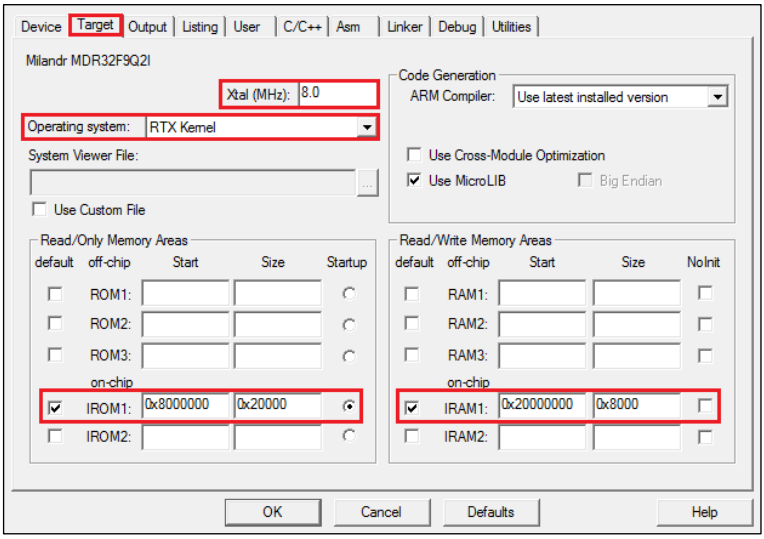


Рисунок 1.10 – Выбор операционной системы, задание тактовой частоты процессора и диапазонов адресов используемой памяти

В нашем случае указывается частота 8 МГц (на такой частоте микроконтроллер запускается, а затем разгоняется до частоты 80 МГц). В микроконтроллерах семейства 1986ВЕ9х доступно 128 Кбайт (0x20000 байт) встроенной флеш-памяти программ (IROM), расположенной по адресам, начиная с 0x8000000, и 32 Кбайт (0x8000 байт) встроенной оперативной памяти данных (IRAM), расположенной по адресам, начиная с 0x20000000. Эти настройки памяти и задаются здесь.

В наших проектах будет использоваться ОСРВ RTX. Чтобы задействовать ее в проекте, выбираем в списке *Operation system* значение *RTX Kernel*. Это позволит подключить к проекту необходимые модули операционной системы, а также сделает доступным специальный монитор отладки ОСРВ RTX.

Далее откроем вкладку *Output* (рисунок 1.11). Тут можно выбрать каталог, в котором будут размещаться выходные файлы проекта, разрешить или запретить создание HEX-файла, а также указать его имя.

Теперь перейдем на вкладку *C/C++* (рисунок 1.12). Здесь задаются настройки компилятора языка C/C++. Нас в основном может заинтересовать уровень оптимизации и пути к библиотекам.

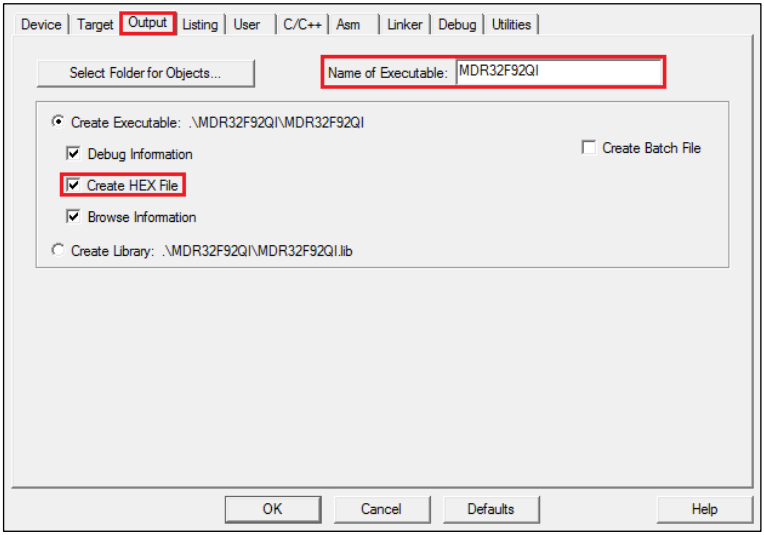


Рисунок 1.11 – Настройка параметров HEX-файла

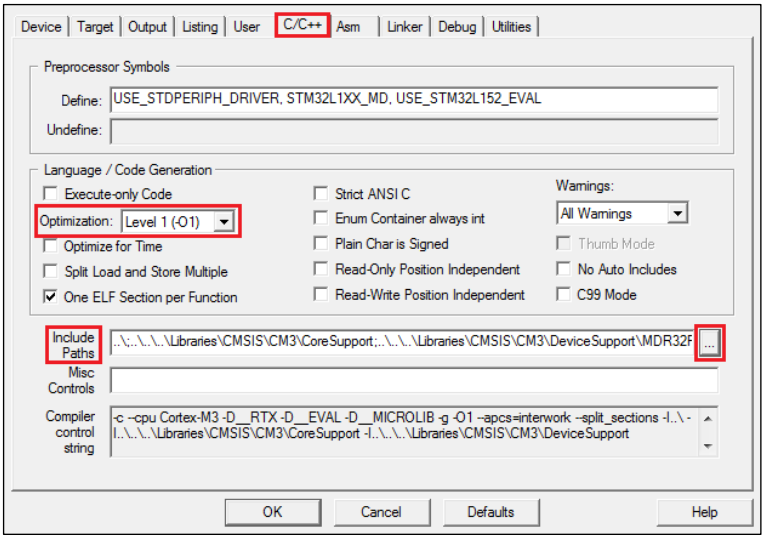


Рисунок 1.12 – Настройки компилятора C/C++

Высокий уровень оптимизации, с одной стороны, позволит создать более компактную и быструю программу. С другой стороны, он может привести к ошибкам при компиляции сложных проектов. Поэтому, если нет явной потребности в оптимизации, рекомендуется выбрать для нее нулевой *Level 0 (-O0)* либо первый уровень *Level 1 (-O1)*.

Пути к библиотекам указываются в поле *Include Paths*. Тут важно указать пути к каталогам, в которых лежат заголовочные файлы библиотек и иных модулей, задействованных в проекте. Для удобства выбора таких путей можно воспользоваться конструктором (рисунок 1.13). По этим путям будет производиться поиск заголовков, подключаемых директивой `#include`.

CMSIS (Cortex Microcontroller Software Interface Standard) – это стандартная библиотека для всех устройств с ядром ARM Cortex. Ее использование значительно упрощает процесс переноса кода с одного вида микроконтроллеров на другие.

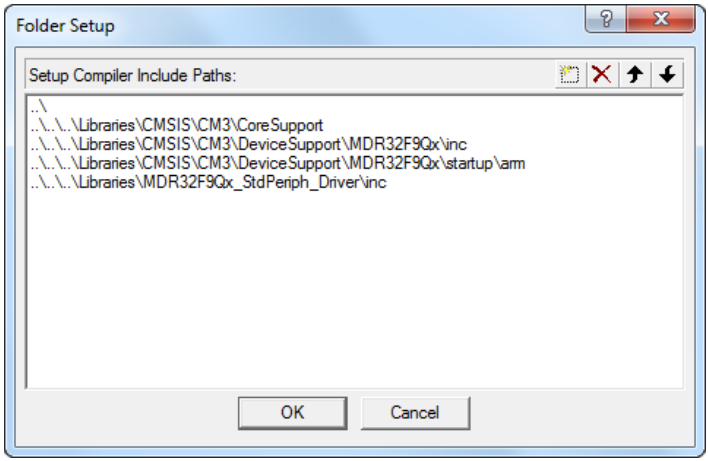


Рисунок 1.13 – Задание путей к заголовочным файлам

Теперь перейдем на вкладку *Debug* (рисунок 1.14). На ней главным образом выбирается тип программатора. В нашем случае это будет *J-LINK / J-Trace Cortex*. Пусть вас не смущает это название. Используемый нами программатор MT-Link является полным функциональным аналогом программатора J-Link. Здесь же можно выбрать и режим эмуляции микроконтроллера с помощью радиокнопки *Use Simulator*. В этом режиме можно отлаживать программы без использования отладочной платы и программатора. К счастью, нам не придется заниматься этим неинтересным и неэффективным способом отладки программ.

Также важно задать некоторые опции, нажав на кнопку *Settings*. В открывшемся окне (рисунок 1.15) на вкладке *Debug* следует выбрать тип интерфейса для отладки и программирования с помощью выпадающего списка *Port*. Вариантов тут два: *SW* и *JTAG*. В нашем случае выбирается *SW*. Частоту программатора выберем равной 1 МГц. Если к компьютеру подключено сразу несколько программаторов, а такое нередко бывает при отладке сложных многопроцессорных систем, следует выбрать из них нужный с помощью списка *SN*. Если программатор всего один, то делать это ни к чему. В списке *SW Device* показывается информация о найденном программатором микроконтроллере.

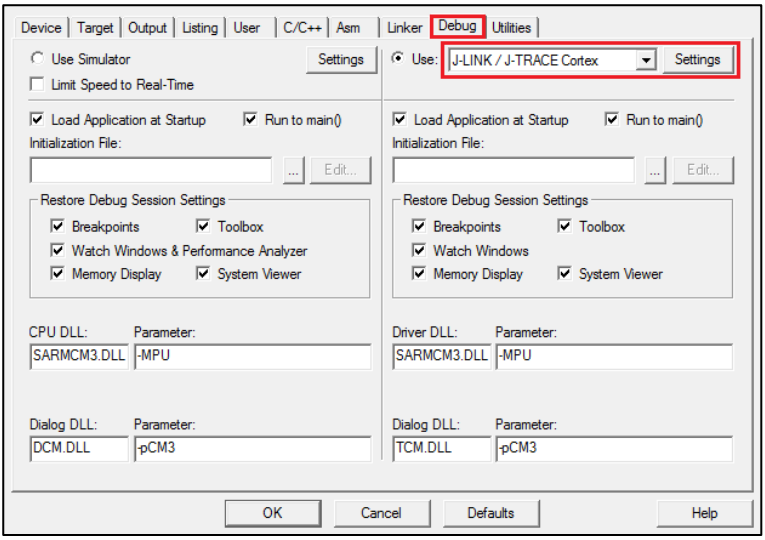


Рисунок 1.14 – Выбор программатора

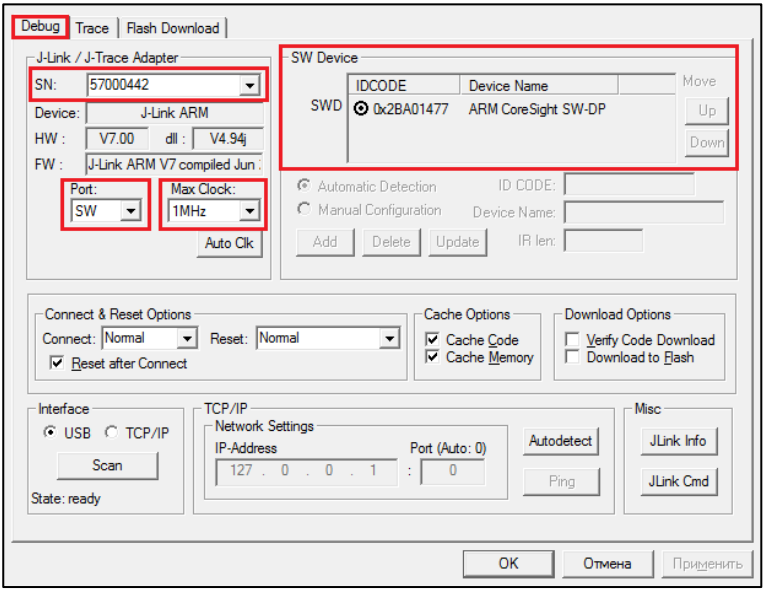


Рисунок 1.15 – Настройка программатора

Вообще на вкладку *Debug* полезно заглянуть в случае, если процесс прошивки не заладился. Если среда Keil «не видит» программатор, то список *SN* будет пуст. Если программатор «не видит» микроконтроллер, то пустым окажется список *SW Device*.

Наконец, перейдем на вкладку *Flash Download* (рисунок 1.16) и настроим процесс загрузки с помощью флажков и радиокнопок в группе *Download Function*.

– *Erase Full Chip* означает стирание всей флеш-памяти;
– *Erase Sectors* – стирание отдельных секторов;
– *Do not Erase* – не стирать ничего;
– *Program* означает, что программа будет загружена в микроконтроллер;
– *Verify* – будет произведена проверка программы;
– *Reset and Run* – по окончании загрузки программа будет автоматически запущена.

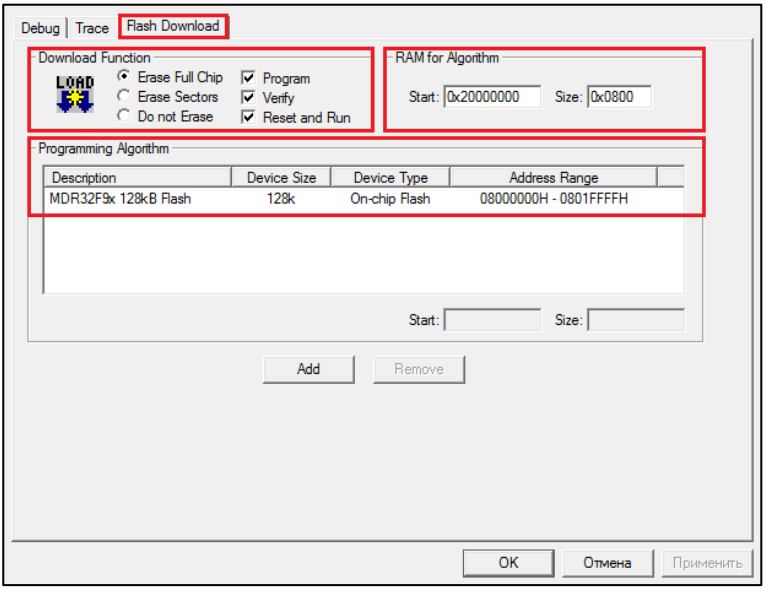


Рисунок 1.16 – Настройка процесса загрузки программы

Также здесь можно задать область оперативной памяти микроконтроллера для работы алгоритма прошивки (поля *Start* и *Size* группы *RAM for Algorithm*). Эти значения зависят от типа микроконтроллера.

Если микроконтроллер прошивался уже много раз, то может случиться, что при проверке правильности загруженной программы возникнут ошибки. В качестве временной меры можно использовать отключение проверки (убрать флаг *Verify*): вполне возможно, что программа все равно будет работать правильно. Но в серьезных проектах это, конечно, недопустимо.

Достаточно часто программисты убирают флаг *Reset and Run*, что позволяет после загрузки программы спокойно перейти в режим отладки и только потом запустить процессор.

Если в поле *Programming Algorithm* отсутствует необходимый алгоритм (*MDR32F9x*), то следует добавить его, нажав кнопку *Add* и выбрав из появившегося списка (рисунок 1.17).

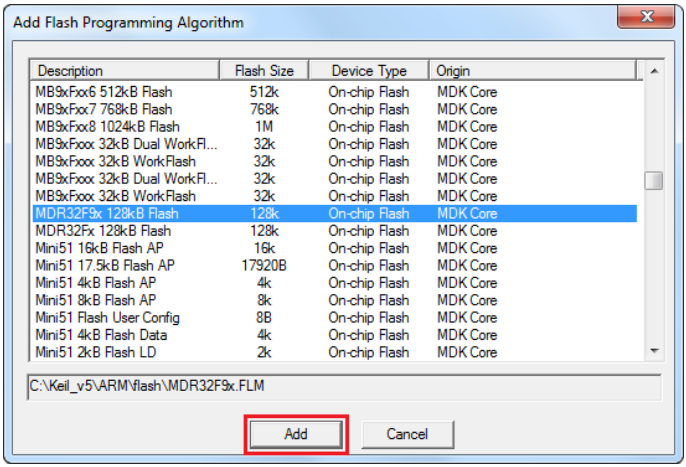


Рисунок 1.17 – Добавление программирующего алгоритма

1.8. Загрузка программы в микроконтроллер

Теперь, когда выполнены все настройки проекта, загрузим («прошьем», «зальем», как говорят программисты) программу в микроконтроллер. Для этого выполним пункт меню *Flash* – *Download* или нажмем кнопку 

на панели инструментов.

Процесс загрузки обычно занимает несколько секунд и, как правило, состоит из следующих этапов:

1. Стирание данных из флеш-памяти микроконтроллера. В этот момент стирается содержимое всей флеш-памяти или указанных в настройках проекта отдельных ее страниц. В нашем случае стираем все. В нижней части главного окна этот процесс иллюстрируется индикатором.

2. Программирование. Данные прошиваются во флеш-память микроконтроллера.

3. Проверка данных. Содержимое флеш-памяти микроконтроллера скачивается обратно на компьютер и сравнивается с тем, что мы пытались прошить. При выявлении несоответствия получаем сообщение об ошибке.

4. Запуск программы на выполнение. Для микроконтроллера генерируется сигнал RESET (сброс), и программа, загруженная в микроконтроллер, начинает свое штатное выполнение.

В окне *Build Output* появляются следующие, понятные из предыдущего пояснения, строки:

```
Full Chip Erase Done.
Programming Done.
Verify OK.
Application running ...
```

Заметим, что часть из перечисленных выше шагов можно отключить с помощью соответствующих настроек, о которых мы говорили ранее (рисунок 1.16).

Если при попытке загрузки появится окно с ошибкой (рисунок 1.18), то это, скорее всего, означает, что отладочная плата не подключена к компьютеру (среда Keil «не видит» программатор). Проверьте соединительный кабель между программатором и компьютером. Иногда может потребоваться переткнуть кабель в другой USB-разъем компьютера, или отключить кабель и подключить вновь.

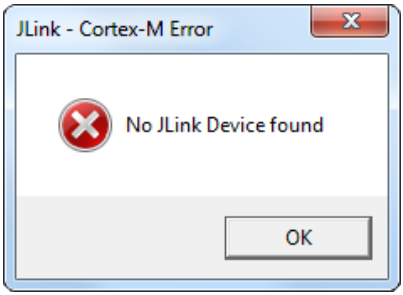


Рисунок 1.18 – Ошибка при загрузке программы в микроконтроллер (среда Keil «не видит» программатор)

Порой также возникает ситуация, когда программатор «не видит» микроконтроллер. В этом случае при попытке прошить микроконтроллер появится сообщение, показанное на рисунке 1.19. Причины такой неполадки могут быть следующие:

1. Шлейф программатора не подключен к разъему JTAG-B.

2. Неправильно выставлены переключатели выбора режима загрузки (рисунок 1.2, позиция 4). В этом случае необходимо выставить переключатели согласно маркировке на отладочной плате.

3. Отсутствует драйвер для программатора. Инструкция по его установке находится в разделе 1.5.

4. В микроконтроллер загружена программа, конфигурирующая выводы PD0 – PD4 как выходы, т.е. JTAG-B заблокирован программой. Для решения этой проблемы переставьте шлейф программатора на разъем JTAG-A и выставьте переключатели выбора режима загрузки в соответствующее положение. Если программой заблокирован и JTAG-A, то микроконтроллер больше не может быть прошит с использованием имеющегося оборудования и подлежит замене.

5. Микроконтроллер неисправен или отсутствует в слоте. Неполадка, очевидно, может быть устранена путем установки исправного микроконтроллера (при его наличии).

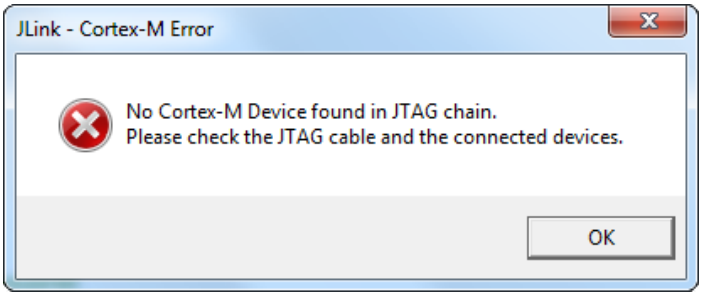


Рисунок 1.19 – Ошибка при загрузке программы в микроконтроллер (программатор «не видит» микроконтроллер)

Итак, прошивка успешно выполнена – теперь микроконтроллер работает самостоятельно, среда Keil ему больше не нужна. Можно отключить программатор от платы – программа сможет работать автономно. Однако **перед отключением программатора не забудьте сначала выключить питание платы**, отсоединив от нее шнур блока питания.

1.9. Внутрисхемная отладка программы

Далее рассмотрим процесс внутрисхемной отладки программы микроконтроллера. Благодаря отладочному интерфейсу SWD, доступному в программаторе MT-Link, этот процесс мало отличается от обычной пошаговой отладки программ в современных средах программирования общего назначения, например, MS Visual Studio.

Для входа в этот режим выполним пункт меню *Debug – Start/Stop Debug Session* или нажмем кнопку на панели инструментов. Главное окно программы несколько изменится (рисунок 1.20). В левой его части будет отображаться состояние регистров микроконтроллера (окно *Register*), вверху появится окно дизассемблера (*Disassebly*). В правой нижней части разместится окно *Call Stack – Locals*, в котором показана последовательность вложенных вызовов функций. В средней части окна остаются вкладки с исходным текстом программы. Но на них теперь дополнительно показывается процесс пошаговой отладки.

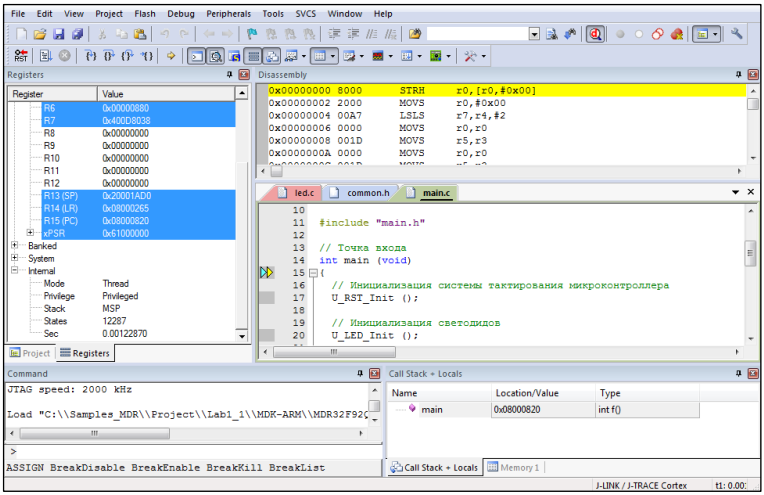


Рисунок 1.20 – Главное окно во время отладки программы

Сейчас программа еще не начала свое выполнение. Если выполнить пункт меню *Debug – Run* (кнопка), то программа будет выполняться до точки останова или непрерывно, если таких точек нет или они недостижимы.

Чтобы приостановить выполнение программы, выполним пункт меню *Debug – Stop* (кнопка). Программа приостановится в самом неожиданном месте. При этом автоматически откроется вкладка с кодом, на котором произошла остановка. Оператор, на котором остановилось выполнение, будет помечен маркером . Чтобы продолжить выполнение нужно выбрать *Debug – Run*.

Для выхода из режима отладки и возврата к обычному редактированию программы следует снова выполнить пункт *Debug – Start/Stop Debug Session.* Иногда перед этим может потребоваться приостановить выполнение программы (*Debug – Stop).*

Для создания точки останова следует установить курсор на требуемый оператор и нажать *F9*. То же самое доступно через меню *Debug – Insert/Remove Breakpoint*. Удобно также ставить точки останова, щелкая по полю слева от номера соответствующей строки. Точка останова помечается красным кружком (рисунок 1.21).

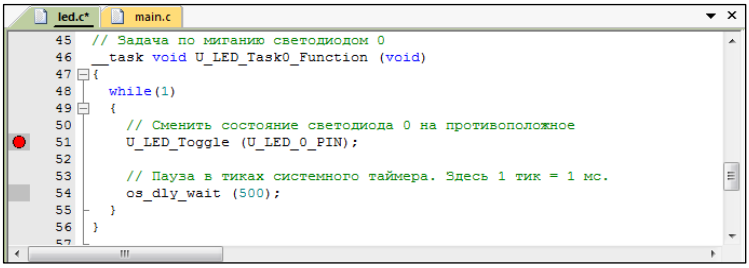


Рисунок 1.21 – Создание точек останова

Попытка повторного создания точки останова на строке, где она уже есть, приводит к удалению ранее назначенной точки. Можно одним разом удалить все ранее созданные точки останова, выполнив пункт меню *Debug – Kill All Breakpoints*. Также можно временно отключить все точки останова: *Debug – Disable All Breakpoints*. Включать их потом придется только по отдельности: *Debug – Enable/Disable Breakpoint*.

Для тренировки поставим точку останова, как показано на рисунке 1.21. Начнем процесс отладки, если это еще не сделано (*Debug – Start/Stop Debug Session*). Выберем вкладку *Project* вместо *Registers* в левой части окна и откроем модуль `led.c`. Найдем строку `U_LED_Toggle (U_LED_0_PIN)` в функции `U_LED_Task0_Function` и создадим на ней точку останова. Запустим программу (*Debug – Run*). Вскоре программа остановится на том месте, где мы задали точку останова (рисунок 1.22). Нажимая кнопку *F5*, будем запускать программу повторно, наблюдая, как через мгновение после нажатия программа вновь остановится на той же строке. При этом красный светодиод LED0 на плате будет то загораться, то гаснуть.

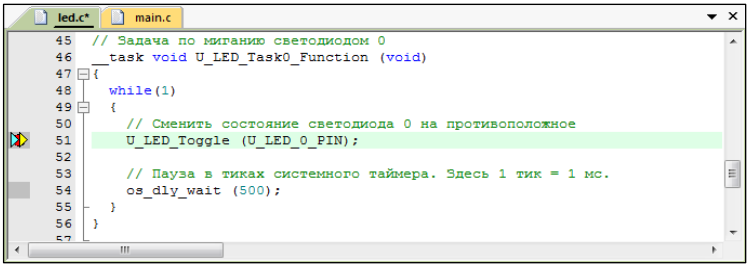


Рисунок 1.22 – Использование точек останова

Так происходит потому, что в функции `U_LED0_Task_Function` организован бесконечный цикл, в котором после паузы в 500 мс (миллисекунд) состояние светодиода LED0 меняется на противоположное. В штатном режиме работы это выглядит как мигание светодиода.

Для пошагового выполнения программы можно использовать следующие действия, хорошо известные программистам и доступные в любой современной системе программирования:

– *Debug* – *Step* () – выполнить оператор с заходом внутрь функции;

– *Debug* – *Step Over* () – выполнить оператор без захода внутрь функции;

– *Debug* – *Step Out* () – выполнить код до выхода из функции.

К примеру, выполним *Debug* – *Step* и попадем внутрь функции `U_LED_Toggle` (рисунок 1.23).

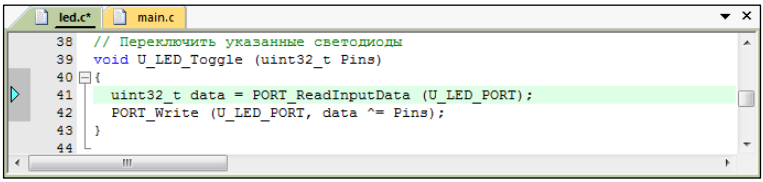


Рисунок 1.23 – Пошаговая отладка

Если снова выполним *Debug* – *Step*, то попадем внутрь библиотечной функции `PORT_ReadInputData` (рисунок 1.24).

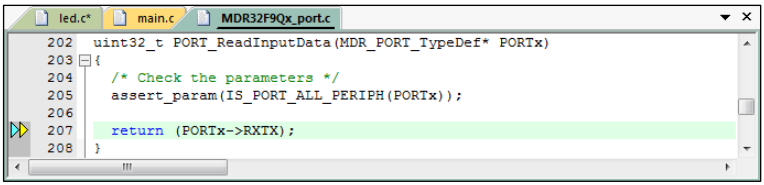


Рисунок 1.24 – Продолжение пошаговой отладки

Выполним *Debug* – *Run to Cursor Line*, чтобы дойти до конца этой функции (хотя он и так близок), а затем *Debug* – *Step Out* для возврата из нее.

Таким образом, на любом шаге можно просматривать значения переменных, находящихся в области видимости, регистров, участков памяти. Подробней об этом будем говорить в последующих разделах.

Задание

Не забудьте выполнить резервное копирование проектов, описанное в разделе 1.1. Это необходимо, чтобы **не портить исходные проекты**.

1. В проекте `Lab1_1` сделайте так, чтобы один светодиод мигал с частотой 4 Гц, а второй – с частотой 0,5 Гц.

Пояснение. Для этого ознакомьтесь с содержанием модуля `led.c` и его заголовка `led.h`. Обратите внимание на программную реализацию мигания, понятие частоты и ее зависимости от периода сигнала.

2. В проекте `Lab1_2` нужно вывести на индикатор бегущую строку с любым словом или словосочетанием на русском языке, состоящим из количества символов в диапазоне от 8 до 16. Строка должна находиться в самом низу дисплея и двигаться со скоростью в два раза большей, чем начальная.

Пояснение. Для этого ознакомьтесь с содержимым модуля `lcd.c`. В заголовке `mlt_font.h` вы найдете кодировку русских букв. Там же приведены изображения всех символов шрифта.

Контрольные вопросы

1. Как подключить отладочную плату к компьютеру?
2. Какие основные элементы имеются на отладочной плате?
3. Как подготовить проект к загрузке в микроконтроллер, и что при этом происходит?
4. Как загрузить программу в микроконтроллер, и что при этом происходит?
5. Как запустить процесс отладки программы?
6. Как поставить точку останова?
7. Какие действия используются при пошаговой отладке?
8. Как называется микроконтроллер, который применялся в работе?
9. Сколько флеш-памяти программ и сколько оперативной памяти данных доступно в микроконтроллере, используемом в работе?
10. Что такое HEX-файл?

Глава 2

Линии ввода-вывода общего назначения

Цель работы: получение навыков работы с портами ввода-вывода общего назначения, светодиодами и механическими кнопками.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- периферийный модуль;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- примеры кода программ.

2.1. Подготовка к работе

В данной работе и нескольких следующих будет использоваться так называемый периферийный модуль, изображенный на рисунке 2.1. Модуль представляет собой небольшую плату на металлических стойках. Устройство платы может быть представлено в виде трех функциональных областей:

- 4 разноцветных светодиода (красный, желтый, зеленый, синий);
- потенциометр 20 КОм;
- лампа накаливания, подключенная через полевой транзистор.

На периферийном модуле и отладочной плате располагаются штыревые разъемы, которые могут быть использованы для их соединения с помощью специальных перемычек, показанных на рисунке 2.2.

Отключите питание отладочной платы, если оно было подключено, и с помощью перемычек подключите к ней периферийный модуль согласно таблице 2.1.

Рисунок 2.1 – Периферийный модуль

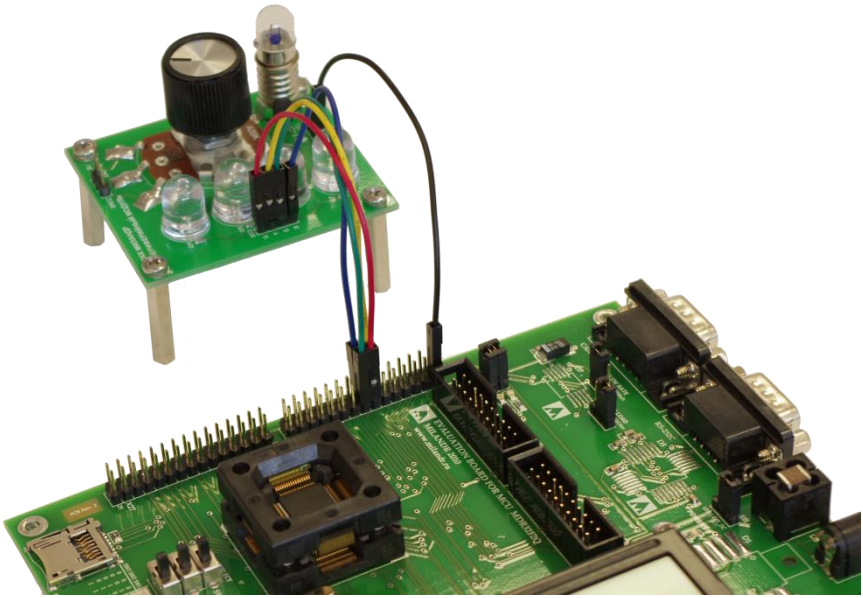


Рисунок 2.2 – Подключение периферийного модуля к отладочной плате

Таблица 2.1 – Подключение периферийного модуля к отладочной плате

№ п/п	Цвет провода	Периферийный модуль		Отладочная плата	
		Имя штыря	Имя разъема	Имя штыря	Имя разъема
1	черный	GND1	XP1	1	X26
2	красный	R	XP3	13	X26
3	желтый	Y	XP3	14	X26
4	зеленый	G	XP3	15	X26
5	синий	B	XP3	16	X26

Следует отметить, что цвета проводов, конечно, не имеют принципиального значения. Однако автор рекомендует придерживаться предложенного варианта для удобства восприятия схемы.

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab2_1\MDK-ARM`. Подключите к отладочной плате программатор и питание (подробности в разделе 1.5). Проверьте правильность настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

2.2. Описание проектов

В примере `Lab2_1` показано, как работать со светодиодами. По очереди загораются красный, желтый, зеленый и синий светодиоды, подключенные к отладочной плате.

В примере `Lab2_2` показано, как использовать порты ввода-вывода совместно с механическими кнопками. Программа работает следующим образом: на ЖКИ появляется надпись «Пункт 0». При нажатии на кнопку «UP» появится надпись «Пункт 1», при следующем нажатии – «Пункт 2», а затем – «Пункт 3». Получается как бы небольшое меню, листаемое нажатием кнопки. После «Пункт 3» вновь, по кругу, будет «Пункт 0». Синий светодиод мигает, напоминая, что система работает.

2.3. Порты ввода-вывода общего назначения

Почти все выводы микросхемы K1986BE92QI представляют собой цифровые линии ввода или вывода. Каждую такую линию можно программным путем сконфигурировать как цифровой вход, либо цифровой выход, и использовать для взаимодействия с внешними схемами. Для удобства использования линии ввода-вывода объединены в порты по 16 линий. Такие порты называют портами ввода-вывода общего назначения.

В англоязычной литературе линии ввода-вывода принято именовать термином GPIO – General-Purpose Input/Output.

К линиям, сконфигурированным как цифровые входы, подключают механические кнопки, выключатели, контакты реле и т.д. Тогда микроконтроллер сможет, например, «узнать», что кнопка нажата. В целом, с помощью таких линий микроконтроллер получает информацию от подключенных к нему устройств.

Линии, сконфигурированные как цифровые выходы, позволяют выдавать управляющие сигналы для подключенных к микроконтроллеру устройств. Таким сигналом через соответствующую схему можно запустить электродвигатель, включить электромагнитное реле или лампу накаливания и т.д.

Каждый цифровой выход у микроконтроллера K1986BE92QI может выдавать ток до 6 мА. Это позволяет, например, подключить к такому выходу яркий светодиод, который будет зажигаться при выдаче соответствующего сигнала.

Всего у микросхемы K1986BE92QI, помещенной в корпус LQFP64, есть 43 линии ввода-вывода, объединенные в 6 портов, как показано в таблице 2.2.

Таблица 2.2 – Порты и линии ввода-вывода микроконтроллера K1986BE92QI

Наименование порта	Количество линий	Наименование линий
PORTA	8	PA0…PA7
PORTB	11	PB0…PB10
PORTC	3	PC0…PC2
PORTD	8	PD0…PD7
PORTE	6	PE0…PE3, PE6…PE7
PORTF	7	PF0…PF6

Разное и отличающееся от шестнадцати количество линий в портах объясняется тем, что в микроконтроллере K1986BE92QI не все линии ввода-вывода реально разведены на ножки микросхемы. Например, в микросхеме 1986BE91T, имеющей 132 ножки, доступны все 96 линий ввода-вывода (6 портов по 16 линий).

На рисунке 2.3 показано, как расположены выводы у микросхемы K1986BE92QI.

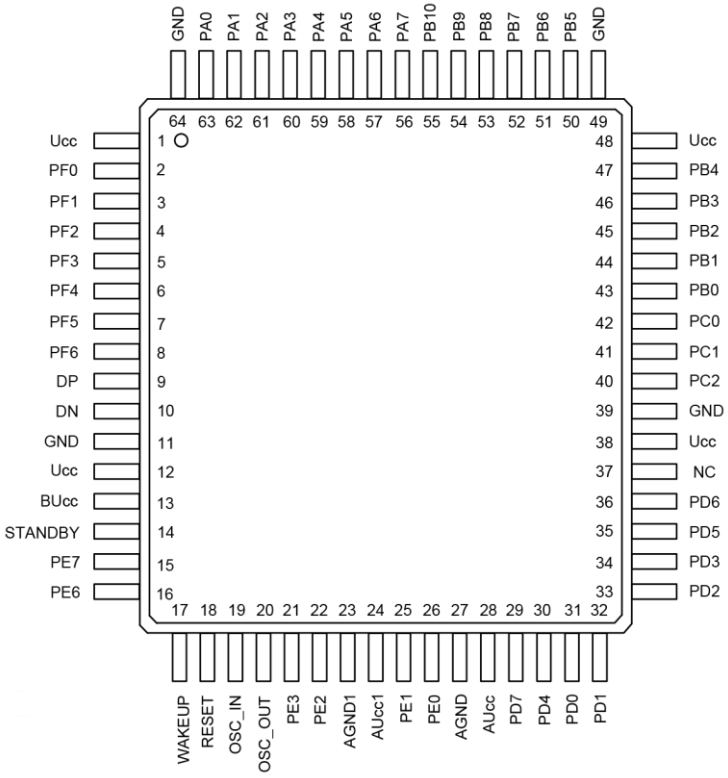


Рисунок 2.3 – Расположение выводов микросхемы K1986BE92QI в корпусе LQFP64

Если касаться внутреннего устройства отдельной линии ввода-вывода, то стоит отметить, что на входе каждой из них имеется два быстродействующих диода Шотки: один подключен к земле, второй –

к плюсу питания. Это нужно для того, чтобы защитить линию от чрезмерного напряжения, которое может появиться на входе в результате разных причин: ошибки при разработке схемы, выхода части схемы из строя, статического электричества и т.п. Такой нехитрый подход позволяет значительно повысить долговечность микросхемы.

Кроме того, на каждой линии расположены так называемые резисторы подтяжки – Pull Up и Pull Down, управляемые транзисторными ключами.

Резистор Pull Up позволяет подтянуть (т.е. подключить) линию к плюсу питания (цепь Ucc). С помощью же резистора Pull Down удается подтянуть линию к земле. Резисторы подтяжки часто требуются при разработке схемы устройства. Наличие таких резисторов непосредственно в микроконтроллере значительно снижает размеры печатной платы разрабатываемого устройства, уменьшает количество компонентов, число паек и стоимость изделия в целом. Однако некоторые разработчики (автор в их числе) с осторожностью используют их, так как сопротивление этих резисторов плохо нормируется, а сами резисторы сравнительно часто выходят из строя. Надежней бывает установить внешний резистор.

Поскольку вопросы схемотехники выходят за рамки нашего курса, будем считать, что особенности настройки резисторов подтяжки и типы выходов для каждого конкретного случая должен сообщить программисту инженер-схемотехник. В лабораторных работах резисторы подтяжки использовать не будем.

2.4. Конфигурирование линий ввода-вывода

Для того чтобы начать использовать линии ввода-вывода, нужно предварительно сконфигурировать их соответствующим образом. Рассмотрим, как это делается.

На самом низком уровне работа с портами ввода-вывода (да и со всеми другими периферийными устройствами) осуществляется с помощью специальных регистров микроконтроллера. Эти регистры доступны как ячейки памяти, расположенные по определенным адресам. Зная эти адреса (они описаны в документации на микроконтроллер), можно записывать в регистры определенные значения, задавая требуемую конфигурацию. Через другие регистры можно получать данные от периферийных устройств.

Если всегда поступать таким образом, то на написание и отладку программного обеспечения уйдет весьма много времени, а программисту придется сосредоточиться не на решении конкретной прикладной задачи, а на «знакомстве» с многочисленными регистрами. Для облегчения труда программиста многие разработчики микроконтроллеров, в том числе и АО «ПКК Миландр», предоставляют специальные библиотеки функций для работы с периферийными устройствами. Для микроконтроллера K1986BE92QI, как и для других микроконтроллеров семейства 1986BE9x, подходит библиотека «MDR32F9Qx Standard Peripherals Library», которую мы и будем использовать в лабораторных работах. В этой библиотеке есть функции для работы со всеми устройствами в составе микроконтроллера. При этом программисту не надо напрямую обращаться к регистрам, достаточно вызвать подходящую функцию, что гораздо удобней. Исходный код каждой функции доступен, что позволяет при необходимости посмотреть, как осуществляется работа с периферией. При желании можно и подправить библиотечные функции, что, однако, не приветствуется.

Исходные тексты модулей библиотеки и их заголовки лежат соответственно в следующих каталогах:

```
Samples\Libraries\MDR32F9Qx_StdPeriph_Driver\src
Samples\Libraries\MDR32F9Qx_StdPeriph_Driver\inc
```

Обратимся к среде Keil. Для работы с портами ввода-вывода к проекту должен быть подключен библиотечный модуль `MDR32F9Qx_port.c`, что уже сделано при разработке проекта. Этот модуль вместе с другими библиотечными модулями находится в группе `StdPeriph_Driver`, которая видна в окне *Project* среды Keil µVision. В модуле `MDR32F9Qx_port.c` есть все необходимое для работы с портами.

Для удобства настройки портов ввода-вывода рекомендуется использовать специальную структуру типа `PORT_InitTypeDef`. Вот пример ее описания:

```
PORT_InitTypeDef PortInitStructure;
```

Тип `PORT_InitTypeDef` описан в заголовке `MDR32F9Qx_port.h` и выглядит так:

```
typedef struct
{
   uint16_t PORT_Pin;
   PORT_OE_TypeDef PORT_OE;
   PORT_PULL_UP_TypeDef PORT_PULL_UP;
   PORT_PULL_DOWN_TypeDef PORT_PULL_DOWN;
   PORT_PD_SHM_TypeDef PORT_PD_SHM;
   PORT_PD_TypeDef PORT_PD;
   PORT_GFEN_TypeDef PORT_GFEN;
   PORT_FUNC_TypeDef PORT_FUNC;
   PORT_SPEED_TypeDef PORT_SPEED;
   PORT_MODE_TypeDef PORT_MODE;
} PORT_InitTypeDef;
```

Перед использованием структуры для конфигурации очередного набора линий ввода-вывода желательно инициализировать ее следующим образом:

```
PORT_StructInit (&PortInitStructure);
```

Это позволит избежать возможных ошибок конфигурирования.

Далее рассмотрим назначения полей этой структуры.

В поле `PORT_Pin` типа `uint16_t` указывают, какие линии подлежат конфигурированию. Каждый бит в этом поле отвечает за отдельную линию ввода-вывода. Эти линии также часто называют выводами или пинами (от англ. pin – вывод, штырь). Линии нумеруются, начиная с нуля. Так, если указать единицу в 4-м разряде, т.е. занести в поле значение `0x0010`, это будет означать, что мы конфигурируем линию 4.

Можно указать единицы сразу в нескольких разрядах, например, во 2-м, 3-м и 5-м. Тогда все настройки будут, соответственно, относится к линиям 2, 3 и 5. Для этого в поле `PORT_Pin` занесем значение `0x002C`.

Чтобы упростить работу с пинами, в модуле `MDR32F9Qx_port.h` предусмотрен ряд констант, описывающих соответствующие линии:

```
#define PORT_Pin_0 0x0001
#define PORT_Pin_1 0x0002
#define PORT_Pin_2 0x0004
#define PORT_Pin_3 0x0008
#define PORT_Pin_4 0x0010
#define PORT_Pin_5 0x0020
#define PORT_Pin_6 0x0040
```

```
#define PORT_Pin_7 0x0080
#define PORT_Pin_8 0x0100
#define PORT_Pin_9 0x0200
#define PORT_Pin_10 0x0400
#define PORT_Pin_11 0x0800
#define PORT_Pin_12 0x1000
#define PORT_Pin_13 0x2000
#define PORT_Pin_14 0x4000
#define PORT_Pin_15 0x8000
#define PORT_Pin_All 0xFFFF
```

Применяя к этим константам операцию поразрядного ИЛИ (символ | – вертикальная палочка), можно легко и понятно указывать требуемый для конфигурации набор линий. Например, если требуется сконфигурировать линии 2, 3 и 5, следует написать:

```
PortInitStructure.PORT_Pin = PORT_Pin_2 | PORT_Pin_3 | PORT_Pin_5;
```

Если требуется сконфигурировать только линию 4, то это выглядит так:

```
PortInitStructure.PORT_Pin = PORT_Pin_4;
```

Константа `PORT_Pin_All` описывает сразу все 16 линий.

В поле `PORT_MODE` типа `PORT_MODE_TypeDef` указывают режим работы выбранных пинов. Здесь возможны два варианта значений:

- `PORT_MODE_ANALOG` – линия является аналоговой;
- `PORT_MODE_DIGITAL` – линия является цифровой.

Режим аналоговой линии выбирается, если требуется работать с АЦП, ЦАП, аналоговым компаратором или внешним низкочастотным кварцевым резонатором. В остальных случаях используется режим цифровой линии.

Если линия сконфигурирована как цифровая, то ее нужно дополнительно сконфигурировать, выбрав одну из возможных функций. Для этого в поле `PORT_FUNC` типа `PORT_FUNC_TypeDef` заносят одно из следующих значений:

- `PORT_FUNC_PORT` – линия используется, как цифровой вход или выход;
- `PORT_FUNC_MAIN` – для линии используется основная функция;
- `PORT_FUNC_ALTER` – для линии используется альтернативная функция;
- `PORT_FUNC_OVERRID` – для линии используется перегруженная функция.

Тут дело в том, что почти каждый пин может выполнять не только функцию простого цифрового входа или выхода. Его можно использовать и для работы с различными периферийными устройствами в составе микроконтроллера: UART, CAN, таймеры/счетчики и др. Причем через один пин может быть доступно до трех различных функций по работе с периферийными устройствами. Их условно называют: основной функцией, альтернативной и перегруженной. На самом деле эти названия никакого приоритета функциям не добавляют.

Функции пинов по работе с периферийными устройствами будут использованы нами в дальнейшем при изучении соответствующих возможностей микроконтроллера, в рамках же текущей работы нас будет интересовать лишь простейшая функция цифрового входа или выхода. Но нужно еще уточнить, чем будет линия – либо входом, либо выходом. Для этого используется поле `PORT_OE` типа `PORT_OE_TypeDef`.

Если требуется сделать пин **цифровым входом**, то напишем:

```
PortInitStructure.PORT_OE = PORT_OE_IN;
```

А если **цифровым выходом**, то:

```
PortInitStructure.PORT_OE = PORT_OE_OUT;
```

В поле `PORT_SPEED` типа `PORT_SPEED_TypeDef` указывают примерную скорость работы линии, т.е. какой частоты сигнал может через нее проходить. Возможны следующие значения:

- `PORT_OUTPUT_OFF` – выход выключен;
- `PORT_SPEED_SLOW` – низкая скорость (фронт порядка 100 нс, частота до 5 МГц);
- `PORT_SPEED_FAST` – высокая скорость (фронт порядка 20 нс, частота до 25 МГц);
- `PORT_SPEED_MAXFAST` – предельно высокая скорость (фронт порядка 10 нс, частота до 50 МГц).

Этот параметр задают для цифровых выходов, он влияет на форму фронтов формируемых импульсов. Если выбрать высокую скорость, то фронты будут крутыми, а если низкую – то пологими (рисунок 2.4). Крутые фронты необходимы на высоких частотах, но они создают широкий спектр

помех, что может неблагоприятно сказаться на работе схемы. Пологие же фронты создают гораздо меньше помех, но на больших частотах неприменимы.

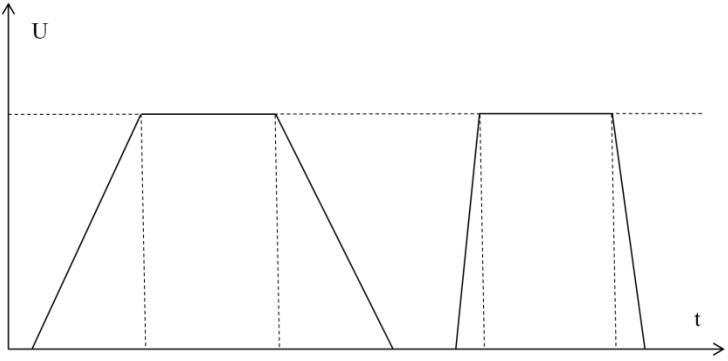


Рисунок 2.4 – Пример пологих и крутых фронтов импульса

В рамках текущей работы со светодиодами нам вполне достаточно низкой скорости:

```
PortInitStructure.PORT_SPEED = PORT_SPEED_SLOW;
```

Поле PORT_PD задает тип цифрового выхода:

– PORT_PD_DRIVER – управляемый драйвер;

– PORT_PD_OPEN – открытый сток.

Не станем сейчас вдаваться в подробности, когда какой тип применяют. Чаще используют управляемый драйвер:

```
PortInitStructure.PORT_PD = PORT_PD_DRIVER;
```

В полях PORT_PULL_UP и PORT_PULL_DOWN задают конфигурацию резисторов подтяжки линии ввода-вывода к цепям питания микроконтроллера, о которых говорилось ранее.

Поле PORT_PULL_UP управляет резистором для подтяжки к плюсу питания. Возможны следующие значения:

– PORT_PULL_UP_OFF – резистор подтяжки не используется;

– PORT_PULL_UP_ON – резистор подтяжки используется.

Поле PORT_PULL_DOWN управляет резистором для подтяжки к земле. Возможны следующие значения:

– PORT_PULL_DOWN_OFF – резистор подтяжки не используется;

– PORT_PULL_DOWN_ON – резистор подтяжки используется.

Как уже отмечалось, в данных работах резисторы подтяжки чаще всего применять не будем, поэтому:

```
PortInitStructure.PORT_PULL_UP = PORT_PULL_UP_OFF;
PortInitStructure.PORT_PULL_DOWN = PORT_PULL_DOWN_OFF;
```

Впрочем, по умолчанию резисторы подтяжки и так отключены.

Поля PORT_PD_SHM и PORT_GFEN задействуют для линии, работающей в режиме входа, триггер Шмидта и цифровой фильтр соответственно. Это позволяет эффективно бороться с помехами, поступающими на вход микроконтроллера вместе с полезным сигналом. Но сейчас об этом говорить рано, и эти поля пока заполнять не будем.

Когда интересующие нас поля структуры заполнены, можно ввести в действие требуемую конфигурацию с помощью функции GPIO_Init:

```
PORT_Init (MDR_PORTC, &PortInitStructure);
```

Первым параметром указывают название порта, для которого производится конфигурация. Предопределенные названия находятся в заголовке MDR32F9Qx_port.h: MDR_PORTA, MDR_PORTB, MDR_PORTC, MDR_PORTD, MDR_PORTE и MDR_PORTF.

Если требуется сконфигурировать пины из нескольких портов, то и функция PORT_Init должна вызываться несколько раз: отдельно для каждого порта.

Также для нормальной работы портов необходимо разрешить их тактирование. Это делается путем вызова специальной функции RST_CLK_PCLKcmd из библиотечного модуля MDR32F9Qx_rst_clk.c:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTB | RST_CLK_PCLK_PORTB |
                 RST_CLK_PCLK_PORTB | RST_CLK_PCLK_PORTB |
                 RST_CLK_PCLK_PORTB | RST_CLK_PCLK_PORTB,
                 ENABLE);
```

В нашем проекте вызов этой функции осуществляется в другой функции – `U_LED_Init()` модуля `led.c`, отвечающего за работу со светодиодами. Не все порты обязательно тактировать, а лишь те, которые реально используются в проекте: лишнее подключение – лишние затраты электроэнергии. Также нужно задавать тактирование и иных периферийных устройств, используемых в разрабатываемом приборе. Вопросы, связанные с тактированием, будут рассмотрены в следующих разделах.

Естественно, конфигурирование линий ввода-вывода должно осуществляться до их первого использования. Обычно это делают до запуска ОСРВ RTX. Но никто не запрещает переконфигурировать любую линию ввода-вывода и в процессе работы устройства. Нередко, например, требуется периодически делать пин то входом, то выходом.

Рассмотрим примеры конфигурирования линий ввода-вывода.

Пример 1. Требуется сконфигурировать линии `PB0`, `PB1`, `PB2` и `PB3`, как цифровой выход для работы с медленными сигналами. Решение:

```
PORT_InitTypeDef PortInitStructure;
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTB, ENABLE);

PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_Pin = PORT_Pin_0 | PORT_Pin_1 |
                             PORT_Pin_2 | PORT_Pin_3;
PortInitStructure.PORT_MODE = PORT_MODE_DIGITAL;
PortInitStructure.PORT_FUNC = PORT_FUNC_PORT;
PortInitStructure.PORT_OE = PORT_OE_OUT;
PortInitStructure. PORT_PD = PORT_PD_DRIVER;
PortInitStructure. PORT_PULL_UP = PORT_PULL_UP_OFF;
PortInitStructure. PORT_PULL_DOWN = PORT_PULL_DOWN_OFF;
PortInitStructure.PORT_SPEED = PORT_SPEED_SLOW;
PORT_Init (MDR_PORTB, &PortInitStructure);
```

Пример 2. Требуется сконфигурировать линии `PB5` и `PB6`, как цифровые входы без резисторов подтяжки. Решение:

```
PORT_InitTypeDef PortInitStructure;
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTB, ENABLE);

PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_Pin = PORT_Pin_5 | PORT_Pin_6;
PortInitStructure.PORT_MODE = PORT_MODE_DIGITAL;
PortInitStructure.PORT_FUNC = PORT_FUNC_PORT;
```

```
PortInitStructure.PORT_OE = PORT_OE_IN;
PortInitStructure.PORT_PULL_UP = PORT_PULL_UP_OFF;
PortInitStructure.PORT_PULL_DOWN = PORT_PULL_DOWN_OFF;
PORT_Init (MDR_PORTB, &PortInitStructure);
```

Заметим, что настройки, касающиеся скорости работы и типа выхода, для цифрового входа ни к чему, поэтому здесь и нет соответствующих строк.

Для имен портов и отдельных линий удобно задавать осмысленные названия-псевдонимы. Делается это, например, так:

```
// Порт светодиодов
#define U_LED_PORT MDR_PORTB
// Линия для красного светодиода
#define U_LED_RED_PIN PORT_Pin_0
// Линия для желтого светодиода
#define U_LED_YELLOW_PIN PORT_Pin_1
// Линия для зеленого светодиода
#define U_LED_GREEN_PIN PORT_Pin_2
// Линия для синего светодиода
#define U_LED_BLUE_PIN PORT_Pin_3
```

Теперь везде, где есть ссылка на порт B в контексте использования светодиодов, можно применять более логичный псевдоним `U_LED_PORT`. А для работы с линией, к которой подключен, к примеру, синий светодиод, – использовать псевдоним `U_LED_BLUE_PIN`. Например:

```
PortInitStructure.PORT_Pin = U_LED_BLUE_PIN;
...
PORT_Init (U_LED_PORT, & PortInitStructure);
```

2.5. Работа с цифровым входом

В плане доступных функций работа с цифровым входом совсем проста: можно прочитать состояние определенного входа или всего порта в целом.

Функция `PORT_ReadInputDataBit()` читает состояние определенного входа. В следующем примере производится чтение состояния входа `PB0`.

```
uint8_t flag;
...
flag = PORT_ReadInputDataBit (MDR_PORTB, PORT_Pin_0);
```

Функция PORT_ReadInputData читает состояние всех входов указанного порта, возвращая 16-битное целое число (по количеству линий в порте):

```
uint16_t in_data;
...
in_data = PORT_ReadInputData (MDR_PORTB);
```

2.6. Работа с цифровым выходом

Посмотрим, как происходит работа с цифровым выходом.

Будем считать, что у нас есть уже сконфигурированный цифровой выход PB0. Требуется выдать на этот выход значение логической единицы. Это можно сделать с помощью функции PORT_SetBits:

```
PORT_SetBits (MDR_PORTB, PORT_Pin_0);
```

Первый параметр – название порта, второй – пины, которые нужно установить в единицу. Естественно, можно указать сразу несколько пинов:

```
PORT_SetBits (MDR_PORTB, PORT_Pin_0 | PORT_Pin_1 | PORT_Pin_2);
```

Если среди указанных пинов попадется цифровой вход, то ничего страшного не произойдет: эта линия будет проигнорирована. Выходы в пределах выбранного порта, которые не указаны при вызове функции, не будут подвергаться какому-либо воздействию.

Для установки на выходе значения логического нуля можно применить функцию PORT_ResetBits, например:

```
PORT_ResetBits (MDR_PORTB, PORT_Pin_0);
```

Функция PORT_WriteBit позволяет установить выбранные выходы в указанное состояние (Bit_RESET – в 0; Bit_SET – в 1). В следующем примере в ноль будут установлены выходы PB0 и PB1, остальные же линии порта B никак не будут затронуты:

```
PORT_WriteBit (MDR_PORTB, PORT_Pin_0 | PORT_Pin_1, Bit_RESET);
```

Функция PORT_Write позволяет просто записать в указанный порт (т.е. во всю совокупность его линий) требуемое значение, заданное 16-битным целым числом. В следующем примере линии PB0 и PB3 будут установлены в единицу, а остальные линии порта B – в ноль:

```
PORT_Write (MDR_PORTB, 0x0009);
```

Таким образом, при вызове функции PORT_Write воздействию подвергаются **все** выходы в пределах выбранного порта. Линии, не являющиеся выходами, естественно, не затрагиваются.

Нередко требуется узнать, в каком состоянии сейчас находится тот или иной цифровой выход. Для этого можно применить уже рассмотренные нами функции PORT_ReadInputDataBit() и PORT_ReadInputData(). Дело в том, что при их вызове в любом случае возвращается состояние линий ввода-вывода независимо от того, вход это или выход.

В следующем примере в переменную flag считывается состояние выхода PB0:

```
uint8_t flag;
...
flag = PORT_ReadInputDataBit (MDR_PORTB, PORT_Pin_0);
```

Если выход установлен в 1, то и переменной flag присвоится значение 1. Если же выход установлен в 0, то и переменной flag присвоится значение 0.

2.7. Особенности работы со светодиодами

Светодиоды широко применяют в современных электронных устройствах для индикации. Они гораздо долговечней и экономичней ламп накаливания. При этом выпускают светодиоды, горящие разными цветами, с разной яркостью, разного размера и формы. Светодиод достаточно ярко горит при сравнительно небольшом токе (от 1 до 20 мА) и напряжении (порядка 1,8 В), что позволяет подключать его непосредственно к выводам микроконтроллера, используя токоограничивающий резистор (иначе ток будет слишком большим: яркости это сильно не добавит, а светодиод долго не проживет, да и вывод микроконтроллера будет перегружен).

На рисунке 2.5 приведен фрагмент принципиальной схемы отладочной платы микроконтроллера K1986BE92QI, на котором показано, как подключаются светодиоды.

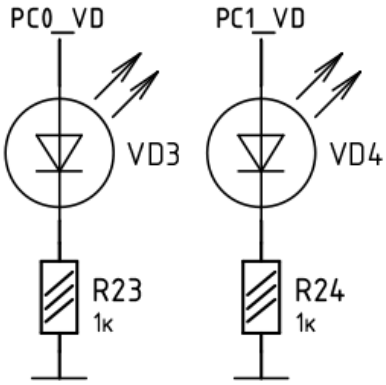


Рисунок 2.5 – Схема подключения светодиодов

Как видно из схемы, светодиод VD3 подключен к линии `PC0`, а светодиод VD4 – к линии `PC1`. Оба светодиода красные.

Теперь для работы со светодиодами нужно инициализировать линии `PC0` и `PC1`, сделав их цифровыми выходами, а затем задавать на этих выходах требуемые значения. Если на выходе `PC0` задать единицу, то загорится светодиод VD3: на аноде светодиода относительно земли, к которой подключен его катод, будет положительное напряжение порядка 1,8 В, достаточное для свечения. Если на выход `PC0` подать ноль, то светодиод VD3 погаснет: между анодом и катодом светодиода будет одинаковое напряжение равное 0 В относительно земли, ток через светодиод не потечет.

К сожалению, на нашей отладочной плате штатные светодиоды присоединены к линиям ввода-вывода, которые также используются в работе с ЖКИ. Это объясняется сравнительно небольшим количеством ножек у микроконтроллера K1986BE92QI – ни одной свободной линии ввода-вывода не остается, все занято какими-либо функциями отладочной платы. Поэтому одновременно работать и с ЖКИ, и с этими светодиодами очень сложно – практически невозможно.

С ЖКИ расставаться жалко, поэтому целесообразно при знакомстве с микроконтроллером и отладке программ применять внешние по отношению к отладочной плате светодиоды. По этой причине мы будем использовать описанный в разделе 2.1 периферийный модуль, фрагмент схемы которого показан на рисунке 2.6. Такое устройство при желании можно элементарно смонтировать самому.

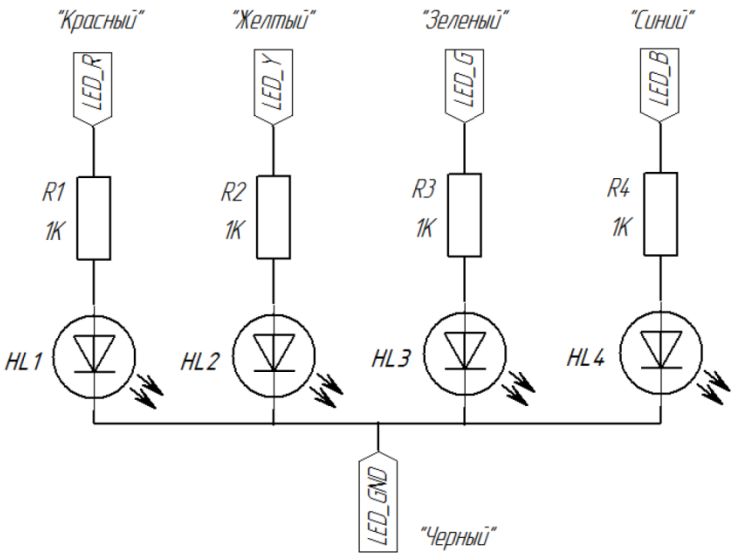


Рисунок 2.6 – Схема модуля светодиодов

Куда подключить светодиоды, если, как мы уже говорили, все линии ввода-вывода уже чем-нибудь заняты? В таких случаях обычно жертвуют теми функциями отладочной платы, которые в данном проекте не нужны или малозначительны. В нашем случае одним из наиболее целесообразных решений будет использование линий `PB0...PB3`, совмещенных с отладочным интерфейсом JTAG-A (см. принципиальную схему отладочной платы). Их, естественно, придется сконфигурировать как цифровые выходы.

Но не забывайте об осторожности! Когда выводы JTAG сконфигурированы как выходы, соответствующий интерфейс JTAG становится недоступным для программатора. Конечно, одновременно два

интерфейса JTAG не нужны, но если сконфигурировать как выходы еще и линии PD0…PD4, относящиеся к JTAG-B, то микроконтроллер будет заблокирован для прошивки. В этом случае придется менять микросхему.

Для повышения наглядности программного кода в модуле led.c описано несколько простых и понятных функций для работы с этими светодиодами. Все они используют ранее рассмотренные функции для работы с портами ввода-вывода:

```
// Потушить указанные светодиоды
void U_LED_Off (uint32_t Pins)
{
   PORT_ResetBits (U_LED_PORT, Pins);
}

// Зажечь указанные светодиоды
void U_LED_On (uint32_t Pins)
{
   PORT_SetBits (U_LED_PORT, Pins);
}

// Переключить указанные светодиоды
void U_LED_Toggle (uint32_t Pins)
{
   uint32_t data = PORT_ReadInputData (U_LED_PORT);
   PORT_Write (U_LED_PORT, data ^= Pins);
}
```

Заметим, что в качестве параметра этих функций указывают имя выхода, к которому подключен требуемый светодиод. Можно при этом использовать псевдонимы U_LED_RED_PIN, U_LED_YELLOW_PIN, U_LED_GREEN_PIN и U_LED_BLUE_PIN, определенные в заголовке led.h

Функция U_LED_Init() рассмотренным ранее способом производит инициализацию выходов, к которым подключены светодиоды. Обратите внимание, в какой момент она вызывается из модуля main.c.

2.8. Особенности работы с механическими кнопками

2.8.1. Принцип работы

Очень часто в составе устройства, использующего микроконтроллер, имеются механические кнопки (а равно и иные механические средства

коммутации: тумблеры, переключатели и т.п.). С их помощью пользователь должен иметь возможность управлять устройством.

Кнопки обычно подключают к линиям ввода-вывода, сконфигурированным как цифровые входы (хотя есть и решения с использованием аналоговых входов, но это – отдельная история).

На рисунке 2.7 приведен фрагмент принципиальной схемы отладочной платы для микроконтроллера K1986BE92QI, на котором показано, как подключены кнопки «UP», «DOWN», «LEFT», «RIGHT» и «SEL» к выводам микроконтроллера. Нас будет интересовать лишь кнопка «UP», подключенная к выводу PB5. Пока кнопка не нажата, вывод PB5 через резистор R11 притянут к +3,3 В, следовательно, на нем будет напряжение логической единицы (те же +3,3 В). Если кнопку нажать, вывод PB5 притянется к цепи питания 0 В, т.е. на PB5 будет напряжение логического нуля.

Как видно из схемы, к линиям ввода-вывода микроконтроллера кнопки подключены не напрямую, а через выключатели S11:1 – S11:5. Это сделано для того, чтобы резисторы R7, R8, R9, R11, R14 и конденсаторы C9, C10, C12, C13, C14 не мешали использованию линий ввода-вывода по иному назначению. Если кнопку используем, то соответствующий выключатель замыкаем, иначе – размыкаем. Как уже было оговорено, в нашей работе потребуется лишь одна кнопка – «UP», поэтому замкнем выключатель S11:1, а остальные выключатели разомкнем. Выключатели S11:1 – S11:5 находятся с обратной стороны платы; они очень маленькие – пальцем не подцепить, поэтому воспользуйтесь чем-нибудь острым, например, шариковой ручкой.

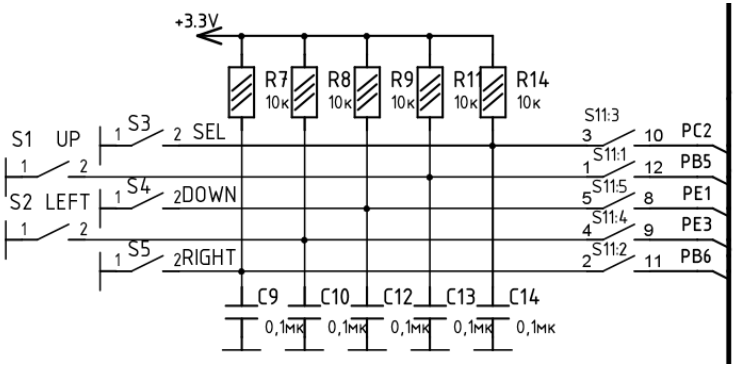


Рисунок 2.7 – Схема подключения механических кнопок

2.8.2. Дребезг контактов

Конденсатор C13 сглаживает помехи на входе PB5, вызванные так называемым дребезгом контактов. Дело в том, что механическая кнопка не сразу уверенно замыкается или размыкается. При нажатии на нее, в течение небольшого периода времени (иногда до 100 мс), контакты кнопки оказываются то замкнутыми, то разомкнутыми. Следовательно, на входе микроконтроллера будет то 0, то 1 (рисунок 2.8). Если не предпринять специальных мер защиты, это может привести к ложным срабатываниям: микроконтроллер может посчитать, что кнопка нажата несколько раз. Конденсатор частично (но не гарантированно) спасает от этого эффекта. Наиболее правильным считается введение специальной программной защиты от этого явления, которую мы рассмотрим.

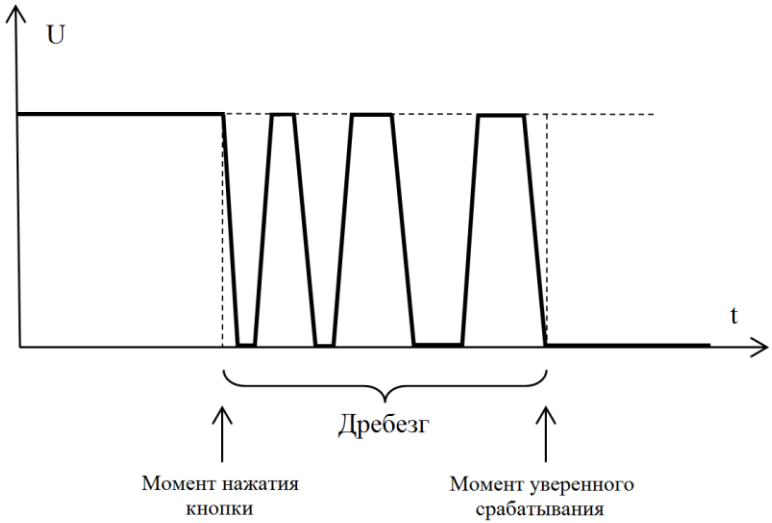


Рисунок 2.8 – Дребезг контакта кнопки

В модуле button.c определены две функции для работы с кнопкой.

Функция U_BTN_Init() производит инициализацию вывода PB5, к которому подключена кнопка «UP», делая его входом.

Функция U_BTN_Read_Button читает состояние входа PB5 и определяет, нажата кнопка (вернет 0) или отпущена (вернет 1).

Как видно из этих простых функций, никакой дополнительной обработки здесь нет. Вся логика работы с кнопкой (листание меню) сосредоточена в модуле menu.c.

В первую очередь, здесь описана функция-задача RTX под названием U_MENU_Task_Function(). Как и в прошлой работе задача создается в функции Main_Task_Init() модуля main.c. Вторая задача отвечает за моргание синего светодиода.

Задача U_MENU_Task_Function() производит периодический опрос кнопки и в случае нажатия производит листание меню. Понажимав несколько раз кнопку, нетрудно заметить, что никаких ложных срабатываний нет. Более того, если очень быстро нажать и отпустить кнопку, система проигнорирует такое действие. Рассмотрим, как это достигается.

2.8.3. Машина состояний

Начиная с этого момента при программировании микроконтроллера мы будем использовать математический аппарат **теории конечных автоматов** или, по-другому, **машины состояния** [12]. С точки зрения задачи по листанию меню наша система может быть описана в виде конечного автомата (по-английски – State Machine – машина состояний), который имеет четыре возможных состояния и переходит между ними в зависимости от действий пользователя по нажатию кнопки и текущего момента времени.

Состояния следующие:

1) ожидание нажатия кнопки;

2) пауза для защиты от дребезга контактов и случайных нажатий;

3) подтверждение нажатия кнопки;

4) ожидание отпускания кнопки.

Переходы между состояниями удобно проиллюстрировать **графом состояний**, изображенном на рисунке 2.9.

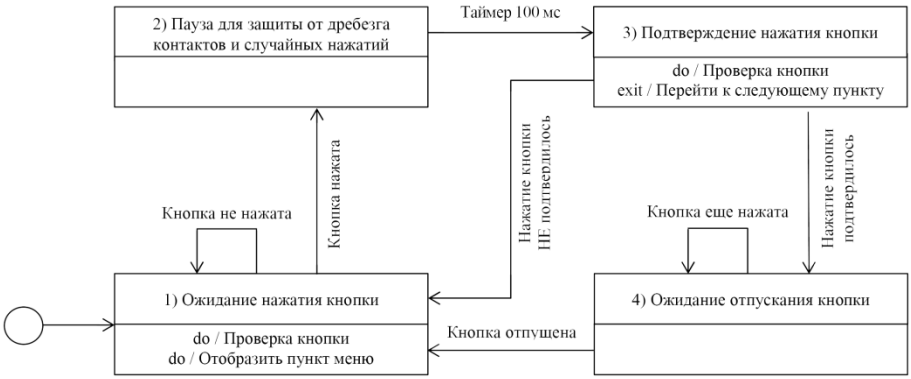


Рисунок 2.9 – Граф состояний

Вершины графа соответствуют состояниям, а линии со стрелками – переходам между состояниями.

В начальный момент автомат находится в состоянии 1, т.е. ждет нажатия кнопки пользователем. Если кнопка не нажата (на входе `PB5` логическая единица), он продолжает оставаться в этом состоянии (петля у вершины 1). При этом каждый раз при заходе в это состояние будет производиться отображение на ЖКИ выбранного пункта меню.

Если пользователь нажмет кнопку, на входе `PB5` появится логический ноль и автомат перейдет в состояние 2. В нем он пробудет небольшое время, достаточное для устранения эффекта дребезга контактов (в данном случае порядка 100 мс), после чего перейдет в состояние 3.

При заходе в состояние 3 будет произведена повторная проверка состояния кнопки. Если кнопка все еще нажата, автомат считает, что нажатие действительно состоялось. Тогда будет произведена смена номера выбранного пункта меню и переход к состоянию 4. Если же кнопка не нажата, то это будет расценено, как попытка ложного срабатывания. Автомат вернется к начальному состоянию 1.

В состоянии 4 автомат дождется отпускания кнопки и лишь тогда вернется к начальному состоянию 1. Это позволит избежать эффекта быстрого неконтролируемого листания меню, если долго держать кнопку нажатой. А так, для листания нужно нажимать и отпускать кнопку в умеренном темпе.

Вот и весь конечный автомат.

Теперь посмотрим, как все это реализовано в программе. Для реализации конечного автомата используется модуль machine.c. Он не то, чтобы стандартный, но использовался автором в том или ином виде во множестве проектов.

Центральной частью модуля является структура конечного автомата, описанная типом TMachine:

```
typedef volatile struct
{
   // Текущее состояние
   // (состояние 0 считается неопределенным – его использовать нельзя)
   int8_t State;
   // Предыдущее состояние
   int8_t Previous;
   // Следующее состояние
   int8_t Next;
   // Таймер-счетчик
   uint32_t Counter;
} TMachine;
```

Для каждого конечного автомата, используемого в проекте, нужно создать отдельный экземпляр такой структуры.

Поле State хранит текущее состояние автомата. Поле Previous – состояние, в котором был автомат перед приходом в текущее состояние. Поле Next – состояние, в которое перейдет автомат, если сработает таймер. Поле Counter – состояние таймера/счетчика, если он установлен.

Перед использованием конечного автомата, его необходимо инициализировать, для этого используется функция U_Machine_Init():

```
void U_Machine_Init (TMachine *machine, // Конечный автомат
                     int8_t start_State // Начальное состояние
                     );
```

В качестве параметров задается указатель *machine на структуру конечного автомата (во всех остальных функциях первый параметр такой же) и номер начального состояния start_State. Заметим, что нельзя в качестве состояния использовать 0 – это значение зарезервировано.

Для смены состояния используется функция U_Machine_Change_State():

```
void U_Machine_Change_State (TMachine *machine,    // Конечный
автомат
                             int8_t newState       // Новое
                             состояние
                             );
```

Параметр `newState` задает новое состояние.

Функция `U_Machine_Stay_Here()` реализует часто встречающееся действие – остаться в текущем состоянии:

```
void U_Machine_Stay_Here (TMachine *machine);
```

Также очень часто нужно бывает проверить: первый ли раз мы зашли в текущее состояние или уже были здесь на предыдущем шаге? Для этого есть функция `U_Machine_Come_From_Another()`. Она вернет 1, если пришли из другого состояния (только что зашли сюда), и 0, если из этого же состояния (т.е. уже были здесь).

```
int8_t U_Machine_Come_From_Another (TMachine *machine);
```

Функция `U_Machine_Set_Timer()` устанавливает таймер, по срабатыванию которого автомат перейдет в состояние `State`:

```
void U_Machine_Set_Timer (TMachine *machine,
                          uint32_t Period,
                          int8_t State
                          );
```

Период времени, через который сработает таймер, задается в параметре `Period` в количестве проверок автомата (число вызовов функции `U_Machine_Test_Timer()`). Функция `U_Machine_Reset_Timer()` сбрасывает ранее установленный таймер:

```
void U_Machine_Reset_Timer (TMachine *machine);
```

И, наконец, функция `U_Machine_Test_Timer()` проверяет факт срабатывания таймера. При каждой такой проверке поле `Counter` в структуре автомата уменьшается на единицу. Как только достигнет нуля, таймер

считается сработавшим, и автомат переводится этой функцией в состояние, указанное в поле Next.

```
void U_Machine_Test_Timer (TMachine *machine);
```

Теперь о том, как же всем этим пользоваться. Обычно конечный автомат работает в бесконечном цикле отдельной задачи ОСРВ. В начале такого цикла с помощью функции RTX os_dly_wait() делается пауза, задающая интервал, через который будет производиться переход между состояниями автомата. Величина этой паузы задает дискретность работы автомата по времени. Ее надо выбирать, исходя из требований решаемой задачи. В нашем случае – 20 мс.

Затем идет вызов функции U_Machine_Test_Timer(), которая проверит, не сработал ли установленный ранее таймер для автомата, и при необходимости переведет автомат в соответствующее состояние.

Далее размещается оператор switch, проверяющий текущее состояние автомата. В зависимости от текущего состояния будет выполнена та или иная ветвь этого оператора. В каждой такой ветви производятся необходимые действия над автоматом. К ним, например, относятся:

- выполнение определенных действий, если мы только что зашли в это состояние; для этого хорошо подходит функция U_Machine_Come_From_Another();
- выполнение определенных действий вне зависимости от того, давно ли мы сюда зашли;
- проверка условий перехода к другим состояниям;
- выполнение определенных действий, перед переходом к другому состоянию;
- переход к другим состояниям (U_Machine_Change_State()) или оставление в этом же состоянии (U_Machine_Stay_Here());
- установка таймера для перехода к определенному состоянию (U_Machine_Set_Timer()).

Далее приведен вид примерной схемы работы с конечным автоматом.

```
// Задача по работе с автоматом
__task void Task (void)
{
   // Описание конечного автомата
   TMachine StateMachine;

   // Инициализация конечного автомата
   U_Machine_Init(&StateMachine, 1);

   // Бесконечный цикл
   while(1)
   {
      // Пауза
      os_dly_wait (20);

      // Проверить таймер конечного автомата
      U_Machine_Test_Timer(&StateMachine);

      // Проверка состояния конечного автомата
      switch (StateMachine.State)
      {
         // Состояние 1
         case 1:

         // Если пришли из другого состояния
         if (U_Machine_Come_From_Another(&StateMachine))
         {
            // Остаться в том же состоянии
            U_Machine_Stay_Here(&StateMachine);
            // Установить таймер для перехода к состоянию 4
            // через 5 периодов проверки конечного автомата
            U_Machine_Set_Timer(&StateMachine, 5, 4);
         }

         // Если произошло требуемое событие, перейти к состоянию
         2
         ...
         U_Machine_Change_State(&StateMachine, 2);
         // Состояние 2
         case 2:
         ...
         // Состояние 3
         case 3:
         ...
         // Состояние N
         case N:
         ...
```

```
            // Если попадет в несуществующее состояние,
            // то инициализировать конечный автомат заново
            default:
            U_Machine_Init (&StateMachine, 1);
            }
        }
}
```

При заходе в состояние 1 производится проверка: заход сюда произведен из другого состояния или из этого же самого. Если из другого, то оставляем автомат здесь и устанавливаем таймер для перехода в состояние 4 через 5 проверок. Это означает, что, если в течение пяти проверок автомат не перейдет в другое состояние, то таймер сработает и переведет автомат в состояние 4.

Далее проверяется, не произошло ли определенное событие. Если произошло, автомат переводится в состояние 2.

Другими словами, автомат пробудет в состоянии 1 до тех пор, пока не произойдет требуемое событие или пока не сработает таймер. В первом случае автомат попадет в состояние 2, а во втором случае – в состояние 4.

Возможных состояний может быть до 255. Если автомат попадет в состояние, которое не предусмотрено (не описано в операторе `switch`), произойдет инициализация автомата, т.е. возврат к состоянию 1.

Именно по такой схеме и с использованием графа состояний, приведенного ранее на рисунке 2.9, и построена работа с меню. Предлагается самостоятельно разобраться с этим по исходным кодам в модуле `menu.c`. Для этого там достаточно комментариев.

На подобных автоматах обычно и строится работа программного обеспечения для микроконтроллера. Правильно описанные автоматы работают очень надежно.

В модуле `menu.c` также есть функция `U_MENU_Prepare_Item()`, которая подготавливает к отображению указанный в качестве параметра пункт меню. Выполняя задание к лабораторной работе, потребуется изменить тело этой функции.

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 2.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. Измените проект `Lab2_1` таким образом, чтобы светодиоды загорались попарно с интервалом в 1 секунду: красный + зеленый, желтый + синий.

2. В проекте `Lab2_2` сделайте вместо четырех пунктов меню шесть пунктов, задав каждому из них какое-нибудь свое индивидуальное название на русском языке.

Пояснение. Количество пунктов меню описывается константой `U_MENU_ITEM_COUNT`, содержащейся в заголовке `menu.h`.

3. Доработайте проект `Lab2_2`, сделав так, чтобы при нажатии кнопки зажигался зеленый светодиод, а при отпускании – гаснул.

Пояснение. Не забудьте добавить в заголовок модуля, который будете дорабатывать, ссылку на заголовок `led.h`.

4. В последнем пункте меню проекта `Lab2_2` реализуйте динамическое отображение надписи «ДА», если синий светодиод в этот момент горит, и «НЕТ» – если не горит.

5. В проекте `Lab2_2` организуйте листание меню без защиты от ложного срабатывания.

Пояснение. Для этого следует организовать переход конечного автомата из состояния 3 сразу в состояние 1, минуя состояние 4.

Контрольные вопросы

1. Что такое линия ввода-вывода?
2. Сколько линий ввода-вывода доступно у микроконтроллера K1986BE92QI?
3. Зачем задают скорость работы цифрового выхода?
4. В каких режимах может работать вывод микроконтроллера K1986BE92QI?
5. Как задать состояние цифрового выхода?
6. Как узнать, в каком состоянии находится цифровой выход?
7. Как прочитать состояние цифрового входа?
8. Какие меры следует предусмотреть для защиты от ложных срабатываний кнопки?
9. Какие есть функции для работы с конечным автоматом? Поясните их назначение.

Глава 3

Аналого-цифровой преобразователь

Цель работы: получение навыков работы с аналого-цифровым преобразователем.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- периферийный модуль;
- мультиметр М-838 или его аналог;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- примеры кода программ.

3.1. Подготовка к работе

Отключите питание отладочной платы, если оно было подключено, и с помощью перемычек подключите к ней периферийный модуль согласно таблице 3.1.

Таблица 3.1 – Подключение периферийного модуля к отладочной плате

№ п/п	Цвет провода	Периферийный модуль		Отладочная плата	
		Имя штыря	Имя разъема	Имя штыря	Имя разъема
1	черный	GND2	XP2	1	X26
2	синий	средний	XP2	11	X26
3	красный	+3,3V	XP2	3	X26

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab3_1\MDK-ARM`. Подключите к отладочной плате программатор и питание (подробности в разделе 1.5). Проверьте правильность

настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

3.2. Описание проектов

В примере `Lab3_1` производится измерение напряжения, снимаемого с выхода потенциометра, в режиме одиночного преобразования по одному каналу с опросом бита окончания преобразования (пояснения далее). На индикатор выводится измеренное значение напряжения в вольтах. Поворачивая ручку потенциометра, можно наблюдать за показаниями на ЖКИ. Если повернуть ручку до упора против часовой стрелки, то на индикаторе должно получиться значение 0 В, если повернуть до упора по часовой стрелке – около 3,3 В.

В примере `Lab3_2` также производится измерение напряжения, снимаемого с выхода потенциометра, но в режиме одиночного преобразования по одному каналу с прерыванием по окончанию преобразования.

В примере `Lab3_3` реализовано измерение температуры корпуса микроконтроллера с помощью встроенного датчика. Результаты измерения выводятся на индикатор в градусах Цельсия.

В примере `Lab3_4` также измеряется температура корпуса, но с применением прямого доступа к памяти (пояснения далее).

В примере `Lab3_5` организовано измерение по двум каналам: напряжение с внешнего потенциометра и температура со встроенного датчика с использованием прямого доступа к памяти.

3.3. Потенциометр

Потенциометр представляет собой переменное сопротивление (переменный резистор), позволяющее создавать так называемый делитель напряжения. На рисунке 3.1 показана принципиальная схема подключения потенциометра к отладочной плате.

Двумя крайними выводами потенциометр подключен к земле (0 В) и напряжению питания (+3,3 В). Третий вывод, обозначенный стрелкой, называют движком. Он является выходом потенциометра. Движок можно перемещать, вращая ручку потенциометра, в любое положение от одного

края до другого. Если движок переместить к выводу +3,3 В, то и на движке тоже будет +3,3 В. Если переместить движок к земле, то на нем будет 0 В. В остальных промежуточных положениях на движке будет получаться напряжение из диапазона от 0 до +3,3 В. Таким образом, потенциометр позволяет создать простейший регулятор напряжения. Нам это потребуется для проведения экспериментов с аналого-цифровым преобразователем.

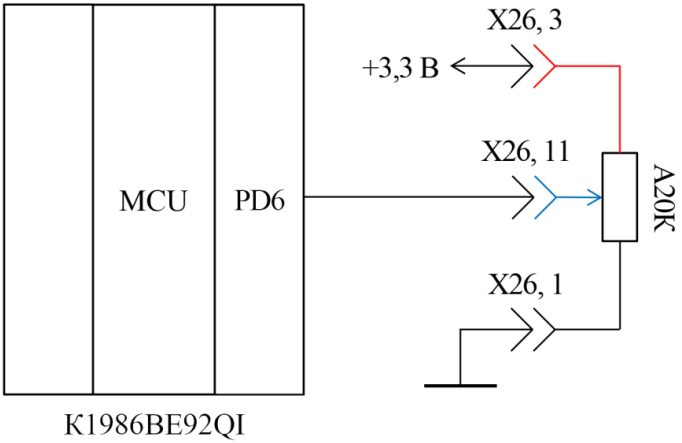


Рисунок 3.1 – Схема подключения потенциометра к отладочной плате

На отладочной плате также имеется встроенный потенциометр, движок которого подключен к выводу PD7 микроконтроллера, а остальные два вывода – к +3,3 В и земле. Он расположен на краю платы между разъемами «ADC» и «USB» (такая маленькая синяя коробочка с винтиком). Этот потенциометр является подстроечным, для его настройки нужна отвертка, что не очень удобно в нашем случае. Поэтому не станем его применять.

Если при выполнении работы внешнего потенциометра не окажется, можно задействовать и внутренний. Но надо будет переставить перемычку «ADC_IN_SEL», расположенную рядом с потенциометром, в положение «TRIM».

При использовании внешнего потенциометра, расположенного на периферийном модуле, для проектов Lab3_1, Lab3_2 и Lab3_5 в заголовке adc.h необходимо закомментировать следующую строку:

```
// #define U_ADC_USE_ONBOARD_POTENTIOMETER
```

Не забудьте это сделать! Иначе на вращение внешнего потенциометра программа не будет реагировать.

3.4. Основы работы с цифровым мультиметром

Мультиметр представляет собой комбинированный измерительный прибор, позволяющий измерять различные электрические параметры. В данной работе используется цифровой мультиметр широко распространенной модели М-838 (рисунок 3.2). Этот прибор позволяет производить следующие измерения:

- постоянное напряжение в диапазонах 0…600 В, 0…200 В, 0…20 В, 0…2000 мВ, 0…200 мВ (работа в режиме вольтметра постоянного тока);
- переменное напряжение в диапазонах 0…600 В, 0…200 В (работа в режиме вольтметра переменного тока);
- постоянный ток в диапазонах 0…10 А, 0…200 мА, 0…20 мА, 0…2000 мкА (работа в режиме амперметра постоянного тока);
- электрическое сопротивление в диапазонах 0…2000 КОм, 0…200 КОм, 0…20 КОм, 0…2000 Ом, 0…200 Ом (работа в режиме омметра);
- температуру в диапазоне -20…+1370 °С (работа в режиме термометра).

Также мультиметр М-838 позволяет «прозванивать» электрические цепи. Термин «прозванивать» означает определение наличия обрыва в электрической цепи, например, в проводе. Если обрыва нет (сопротивление проверяемой цепи не превышает 1 КОм), то мультиметр издает звуковой сигнал. Если же обрыв есть (сопротивление проверяемой цепи более 1 КОм), то прибор будет «молчать».

В рамках нашей работы нам потребуется измерять постоянное напряжение до 3,3 В. Для этого проверьте, чтобы красный щуп был подключен к гнезду «VΩmA», а черный щуп – к гнезду «COM». Поверните ручку мультиметра в положение «20V=» (измерение постоянного напряжения в диапазоне 0…20 В). На индикаторе прибора должна появиться надпись «0.00».

Щупы прислоняют к двум точкам схемы, между которыми нужно померить напряжение. Черный щуп условно считается общим, а красный – сигнальным. Если электрический потенциал на красном щупе будет больше, чем на черном щупе, то прибор покажет напряжение большее нуля.

Напротив, если электрический потенциал на красном щупе будет меньше, чем на черном щупе, то прибор покажет напряжение меньшее нуля. Таким образом, прибор всегда показывает напряжение относительно общего щупа.

Рисунок 3.2 – Внешний вид мультиметра М-838

Измерим напряжение питания, подаваемое на потенциометр. Прислоните черный щуп к выводу потенциометра, к которому подходит черный провод (земля). Красным щупом прикоснитесь к выводу потенциометра, к которому подходит красный провод (+3,3 В). На индикаторе мультиметра появится значение порядка 3,3 В.

Проверьте, что будет, если красным щупом коснуться вывода потенциометра, к которому подходит черный провод (земля), а черным щупом – вывода потенциометра, к которому подходит красный провод (+3,3 В). На индикаторе вы увидите показание порядка минус 3,3 В.

Теперь измерим напряжение, получаемое на выходе потенциометра. Для этого общий щуп подключим к выводу с черным проводом, а сигнальный щуп – к выводу с синим проводом, т.е. к движку потенциометра. На индикаторе прибора появится некоторое значение напряжения. Вращайте ручку потенциометра и наблюдайте за показаниями прибора. Должно получиться плавное изменение напряжения в диапазоне 0…3,3 В.

Приступая к измерению напряжения, надо выбрать соответствующий диапазон измерений. Измеряемое значение напряжения должно укладываться в этот диапазон. Если порядок значений напряжения даже приблизительно неизвестен, сначала выбирают самый длинный диапазон (0…600 В), а затем, проведя пробный эксперимент, уточняют диапазон. Чем короче будет выбран диапазон, тем точнее удастся измерить напряжение.

Если значение измеряемого напряжения превысит верхний предел измерения, то мультиметр покажет переполнение. Из строя он при этом не выйдет. Однако есть допустимый предел измеряемого напряжения, который нельзя превышать. Для прибора М-838 он составляет 600 В. Если подать больше, прибор может выйти из строя.

Следует также помнить, что **безопасным для человека является постоянное напряжение до 36 В**. Большее напряжение может вызвать поражение электрическим током. Поэтому при измерении напряжения более 36 В нельзя касаться частями тела оголенных проводников, в том числе металлических частей измерительных щупов. Необходимо также обращать внимание на техническое состояние щупов мультиметра. На них не должно быть трещин, порезов и прочих видимых повреждений изоляции. **Ни в коем случае нельзя измерять напряжение большее, чем то, на которое рассчитан данный прибор!** Помимо выхода из строя прибора, это может привести к пробою изоляции измерительных щупов ввиду повышения температуры проводов и поражению человека электрическим током.

ВАЖНО! По окончании работы с мультиметром не забудьте выключить его, повернув переключатель в положение «OFF». Если этого не сделать, батарея, питающая прибор, быстро разрядится.

3.5. Понятие аналого-цифрового преобразователя

Аналого-цифровой преобразователь (АЦП или, по-английски, ADC – Analog to Digital Converter) представляет собой устройство, на вход которого подается аналоговый сигнал (произвольное значение напряжения из некоторого диапазона), а на выходе получается цифровой код, пропорциональный поданному на вход напряжению. Таким образом, АЦП позволяет измерять значение напряжения, поданного на его вход, выдавая результат измерений в виде цифрового значения.

АЦП является основой многих измерительных приборов: цифровых вольтметров, цифровых термометров, электронных весов, измерителей давления и т.д. Основой мультиметра М-838, который мы используем в данной работе, тоже является АЦП. С помощью АЦП создается также цифровая звукозаписывающая аппаратура (диктофоны).

АЦП может представлять собой отдельную микросхему, а может также входить в состав микроконтроллера. В микроконтроллер K1986BE92QI как раз интегрирован АЦП, с которым мы и будем работать.

Основными характеристиками АЦП являются разрядность и время преобразования. В микросхеме K1986BE92QI разрядность АЦП составляет 12 бит. Это позволяет АЦП различать до 2^{12} = 4096 различных уровней напряжения, подаваемого на вход.

Время преобразования зависит от тактовой частоты, подаваемой на АЦП. Минимальное время преобразования может составлять 1,95 мкс. Это позволяет производить до 512820 преобразований в секунду. Скорость преобразования можно регулировать, меняя тактовую частоту, подаваемую на АЦП. Но этот вопрос сейчас не будем затрагивать.

В микроконтроллерах семейства 1986BE9x реализовано сразу 2 независимых АЦП – ADC1 и ADC2, они входят в состав контроллера АЦП, структурная схема которого показана на рисунке 3.3.

Во многих микроконтроллерах, в том числе и в микросхеме K1986BE92QI, к одному АЦП подводятся сигналы с нескольких направлений. Это позволяет в разное время выполнять преобразование разных сигналов, т.е. производить несколько различных измерений. Для этого в составе микроконтроллера имеется специальное устройство – аналоговый мультиплексор. На рисунке 3.3 аналоговый мультиплексор обозначен как Analog Matrix.

Входы мультиплексора соединены с линиями ввода-вывода общего назначения, а также со встроенным датчиком температуры VTEMP и датчиком внутреннего опорного напряжения VREF = 1,23 В. Мультиплексор позволяет в каждый момент времени выбрать для каждого АЦП один из каналов измерений. Управление мультиплексором производится программным путем, т.е. программист указывает, какой из каналов сейчас нужен. Выбранный канал измерений посредством мультиплексора соединяется с входом АЦП.

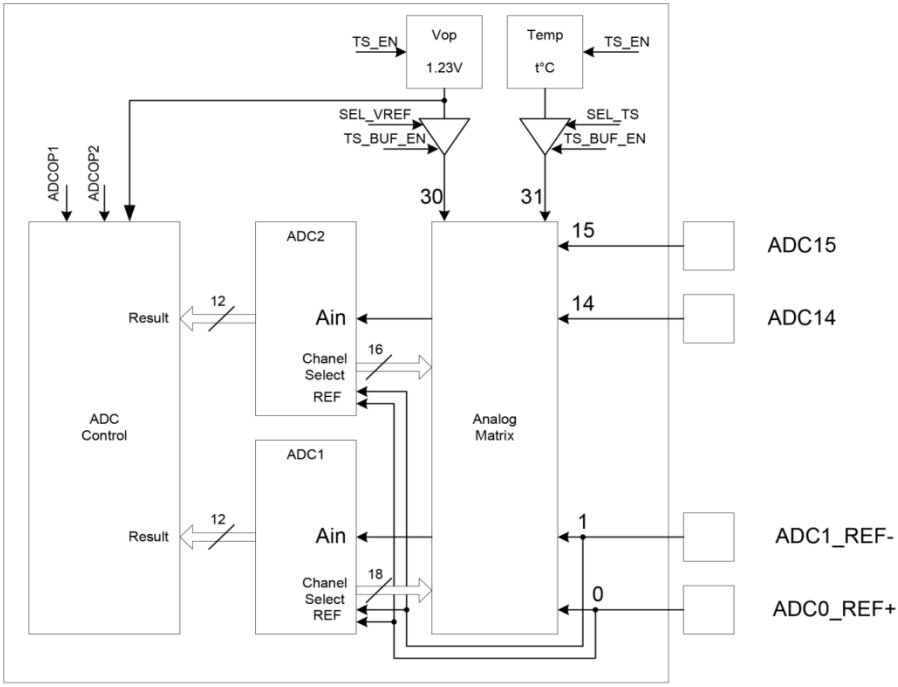


Рисунок 3.3 – Схема контроллера АЦП

По команде программиста АЦП начинает преобразование, по завершении которого результаты преобразования поступают в специальный 32-битный регистр `ADC1_RESULT` для ADC1 или `ADC2_RESULT` для ADC2, откуда их затем можно считать. Как только преобразование завершится, АЦП устанавливает специальный флаг (бит) под названием `FLG_REG_EOCIF` – End of Conversion (конец преобразования), который доступен для чтения в регистрах состояния АЦП – `ADC1_STATUS` и `ADC2_STATUS`. Проверяя значение этого флага, можно определить момент завершения преобразования. Кроме того, при необходимости АЦП может генерировать аппаратное прерывание, сообщающее об окончании преобразования. Об этом будет рассказано несколько позже.

При работе с каждым из двух АЦП в микроконтроллерах семейства 1986ВЕ9х можно использовать следующие основные режимы:

- режим одиночного преобразования по одному каналу с опросом бита окончания преобразования;
- режим одиночного преобразования по одному каналу с прерыванием по окончанию преобразования;
- режим многократного преобразования по одному каналу с использованием прямого доступа к памяти;
- режим многократного преобразования с автоматическим переключением нескольких каналов и использованием прямого доступа к памяти.

Также можно выполнять синхронный запуск сразу двух АЦП, комбинируя его с перечисленными выше режимами. Это позволяет одновременно выполнять аналого-цифровые преобразования сразу для двух разных каналов, что повышает скорость преобразования. Однако в рамках данной работы мы будем использовать лишь один АЦП – ADC1.

3.6. Настройка аналого-цифрового преобразователя

Перед началом использования АЦП, его необходимо инициализировать, как и любое другое периферийное устройство микроконтроллера. Рассмотрим, как это делается на примере проекта `Lab3_1` к данной лабораторной работе. В этом проекте используется режим одиночного преобразования по одному каналу с опросом бита окончания преобразования.

Если проект еще не открыт, откроем его. Затем откроем модуль `adc.c` и рассмотрим функцию `U_ADC_Init`.

Для настройки АЦП используются специальные структуры: `ADC_InitStructure` типа `ADC_InitTypeDef`, и `ADCx_InitStructure` типа `ADCx_InitTypeDef`, в которых предусмотрены все необходимые для этого параметры:

```
ADC_InitTypeDef ADC_InitStructure;
ADCx_InitTypeDef ADCx_InitStructure;
```

Необходимость в двух различных структурах обусловлена тем, что в микроконтроллерах семейства 1986BE9x имеется два АЦП. При этом структура ADC_InitStructure отвечает за общую настройку обоих АЦП, а структура ADCx_InitStructure – за настройку каждого конкретного АЦП: ADC1 или ADC2. Как уже говорилось, для простоты в работе будем использовать лишь один АЦП – ADC1. Однако те же принципы настройки сохраняются и для ADC2.

Для работы с АЦП мы задействуем линию ввода-вывода PD6, к которой будет подключаться выход потенциометра. Поэтому необходимо соответствующим образом сконфигурировать эту линию. Для этого, во-первых, нужно разрешить тактирование АЦП и порта PORTD:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_ADC | RST_CLK_PCLK_PORTD, ENABLE);
```

Во-вторых, нужно сделать вывод PD6 аналоговым:

```
PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_Pin = U_ADC_U_PIN;
PortInitStructure.PORT_MODE = PORT_MODE_ANALOG;
PORT_Init (MDR_PORTD, &PortInitStructure);
```

Константа U_ADC_U_PIN, определенная в заголовке adc.h, соответствует требуемой линии порта PORTD:

```
#define U_ADC_U_PIN PORT_Pin_6
```

Затем следует заполнить структуру ADC_InitStructure необходимыми параметрами:

```
ADC_InitStructure.ADC_SynchronousMode = ADC_SyncMode_Independent;
ADC_InitStructure.ADC_StartDelay = 0;
ADC_InitStructure.ADC_TempSensor = ADC_TEMP_SENSOR_Disable;
ADC_InitStructure.ADC_TempSensorAmplifier =
ADC_TEMP_SENSOR_AMPLIFIER_Disable;
ADC_InitStructure.ADC_TempSensorConversion =
ADC_TEMP_SENSOR_CONVERSION_Disable;
ADC_InitStructure.ADC_IntVRefConversion =
ADC_VREF_CONVERSION_Disable;
ADC_InitStructure.ADC_IntVRefTrimming = 0;
ADC_Init (&ADC_InitStructure);
```

Параметр SynchronousMode указывает, как два имеющихся АЦП будут запускаться: независимо друг от друга (ADC_SyncMode_Independent) или синхронно (ADC_SyncMode_Synchronous). Поскольку мы сейчас будем использовать лишь один АЦП, однозначно выбираем режим независимого запуска.

Параметр ADC_StartDelay позволяет задать небольшую задержку между запусками ADC1 и ADC2 при использовании синхронного запуска. В нашем случае этот параметр ни на что не влияет.

Следующие три параметра – ADC_TempSensor, ADC_TempSensorAmplifier и ADC_TempSensorConversion – управляют работой встроенного в микроконтроллер температурного датчика. Подробней о них мы будем говорить в соответствующем параграфе, а пока лишь заметим, что датчик будет отключен.

Параметры ADC_IntVRefConversion и ADC_IntVRefTrimming управляют внутренним датчиком опорного напряжения. Если параметр ADC_IntVRefConversion установить в значение ADC_VREF_CONVERSION_Enable, то можно использовать датчик опорного напряжения через канал 30 АЦП. С помощью параметра ADC_IntVRefTrimming, задавая значения от 0 до 7, можно в небольших пределах подстроить напряжение этого датчика. Поскольку в наших изысканиях датчик опорного напряжения не понадобится, отключим его, задав параметру ADC_IntVRefConversion значение ADC_VREF_CONVERSION_Disable.

Теперь заполним структуру ADCx_InitStructure:

```
ADCx_StructInit (&ADCx_InitStructure);
ADCx_InitStructure.ADC_ClockSource = ADC_CLOCK_SOURCE_CPU;
ADCx_InitStructure.ADC_Prescaler = ADC_CLK_div_512;
ADCx_InitStructure.ADC_SamplingMode = ADC_SAMPLING_MODE_SINGLE_CONV;
ADCx_InitStructure.ADC_ChannelSwitching = ADC_CH_SWITCHING_Disable;
ADCx_InitStructure.ADC_ChannelNumber = U_ADC_U_CH;
ADCx_InitStructure.ADC_Channels = 0;
ADCx_InitStructure.ADC_DelayGo = 7;
ADCx_InitStructure.ADC_LevelControl = ADC_LEVEL_CONTROL_Disable;
ADCx_InitStructure.ADC_LowLevel = 0;
ADCx_InitStructure.ADC_HighLevel = 0;
ADCx_InitStructure.ADC_VRefSource = ADC_VREF_SOURCE_INTERNAL;
ADCx_InitStructure.ADC_IntVRefSource = ADC_INT_VREF_SOURCE_INEXACT;
ADC1_Init (&ADCx_InitStructure);
```

Параметр ADC_ClockSource задает способ тактирования АЦП. Собственно, здесь может быть всего два варианта: ADC_CLOCK_SOURCE_CPU – тактирование АЦП той же частотой, что и ядро микроконтроллера; и ADC_CLOCK_SOURCE_ADC – тактирование АЦП от отдельного источника, специально предназначенного для АЦП. Для простоты настройки будем использовать первый вариант, что чаще всего и делают на практике. Каждое преобразование требует **не менее 28 тактов.**

С помощью параметра ADC_Prescaler можно задать предделитель частоты тактирования АЦП. Чем выше частота тактирования, тем быстрее будет выполняться преобразование, но при слишком большой частоте может снизиться точность. Поэтому частота тактирования АЦП не должна превышать 14 МГц. Можно использовать значения предделителя из следующего предопределенного ряда:

- ADC_CLK_div_None (предделитель отключен);
- ADC_CLK_div_2 (деление на 2);
- ADC_CLK_div_4 (деление на 4);
- ADC_CLK_div_8;
- ADC_CLK_div_16;
- ADC_CLK_div_32;
- ADC_CLK_div_64;
- ADC_CLK_div_128;
- ADC_CLK_div_256;
- ADC_CLK_div_512;
- ADC_CLK_div_1024;
- ADC_CLK_div_2048;
- ADC_CLK_div_4096;
- ADC_CLK_div_8192;
- ADC_CLK_div_16384;
- ADC_CLK_div_32768.

В нашем случае возьмем предделитель, равный 512, и получим следующую частоту тактирования АЦП:

$$80 \text{ МГц} / 512 = 156{,}25 \text{ КГц}$$

Это позволит выполнять до 156250 / 28 = 5580 преобразований в секунду.

Параметр ADC_SamplingMode задает режим преобразования:

– ADC_SAMPLING_MODE_SINGLE_CONV – одиночное;

– ADC_SAMPLING_MODE_CICLIC_CONV – последовательное.

При одиночном преобразовании после запуска АЦП производится одно преобразование. Для выполнения следующего преобразования нужно вновь запустить АЦП. Сейчас выберем именно этот режим.

При последовательном преобразовании после завершения одного преобразования автоматически запускается новое преобразование. Такой режим обычно используют совместно с прямым доступом к памяти, о чем речь пойдет немного позже.

С помощью параметра ADC_ChannelSwitching может быть включено (ADC_CH_SWITCHING_Enable) или выключено (ADC_CH_SWITCHING_Disable) автоматическое переключение каналов. Смысл состоит в том, что можно заставить микроконтроллер автоматически переключать вход АЦП на другой канал после выполнения очередного преобразования. Это дает возможность задействовать режим многократного преобразования с автоматическим переключением нескольких каналов. Об этом мы будем говорить отдельно. Пока же выключим автоматическое переключение каналов.

Если автоматическое переключение каналов отключено, то нужно указать конкретный канал, с которым АЦП будет работать. Это делается с помощью параметра ADC_ChannelNumber. Можно выбирать следующие каналы:

– ADC_CH_ADC0 – канал 0, порт PD0;

– ADC_CH_ADC1 – канал 1, порт PD1;

– ADC_CH_ADC2 – канал 2, порт PD2;

– ADC_CH_ADC3 – канал 3, порт PD3;

– ADC_CH_ADC4 – канал 4, порт PD4;

– ADC_CH_ADC5 – канал 5, порт PD5;

– ADC_CH_ADC6 – канал 6, порт PD6;

– ADC_CH_ADC7 – канал 7, порт PD7;

– ADC_CH_ADC8...ADC_CH_ADC15 – каналы 8-15 недоступны в микроконтроллере K986BE92QI;

– ADC_CH_INT_VREF – канал датчика опорного напряжения VREF, только для ADC1;
– ADC_CH_TEMP_SENSOR – канал температурного датчика, только для ADC1.

В нашем случае с помощью канала АЦП ADC_CH_ADC6 измеряется напряжение, снимаемое с движка потенциометра, подключенного к отладочной плате. Для повышения наглядности и универсальности в заголовке adc.h описана константа U_ADC_U_CH, которую мы и подставляем вместо номера канала:

```
#define U_ADC_U_CH ADC_CH_ADC6
```

В поле ADC_Channels указывают несколько используемых каналов, если выбран режим с автоматическим переключением каналов. Поскольку мы этот режим сейчас не используем, заносим сюда значение 0.

Также, если выбран режим с автоматическим переключением каналов, можно задать с помощью поля ADC_DelayGo значение задержки перед началом следующего преобразования после завершения предыдущего. Это дает возможность немного подождать пред тем, как начать преобразование по новому выбранному каналу. В некоторых случаях за счет этого точность преобразования может немного повыситься. Задержка задается в тактах ядра микроконтроллера и может составлять от 0 до 7 тактов. По мнению автора, лучше задать ненулевую задержку. Хотя в нашем случае этот параметр ни на что не повлияет, все-таки зададим задержку в 7 тактов.

В микроконтроллерах семейства 1986ВЕ9х при работе с АЦП можно использовать так называемый контроль уровня входного сигнала. Если этот контроль задействован, то микроконтроллер следит, выходит ли напряжение, поступающее на вход АЦП, за пределы диапазона значений от ADC_LowLevel до ADC_HighLevel или нет. Если выходит, то после преобразования в регистре ADCx_STATUS будет установлен специальный флаг Flg_REG_AWOIFEN. В некоторых задачах это оказывается полезной «мелочью», упрощающей и ускоряющей обработку данных. Для включения контроля уровня в поле ADC_LevelControl следует занести значение ADC_LEVEL_CONTROL_Enable, а в полях ADC_LowLevel и ADC_HighLevel указать, соответственно, нижнюю и верхнюю границы допустимого диапазона. В нашем же случае это ни к чему, поэтому в поле

ADC_LevelControl занесем значение ADC_LEVEL_CONTROL_Disable, а в поля ADC_LowLevel и ADC_HighLevel занесем нули.

Еще нужно выбрать для АЦП источник опорного напряжения. Этот источник используется в роли эталона, с которым АЦП сравнивает измеряемое напряжение. Чем точнее и стабильнее от температуры и во времени источник опорного напряжения, тем точнее будет выполняться аналого-цифровое преобразование. Программист может выбирать из двух вариантов:

1) внутренний источник (ADC_VREF_SOURCE_INTERNAL). В качестве источника используется напряжение между выводами AUCC и AGND, т.е. напряжение питания аналоговой части микроконтроллера. Задается константой ADC_VREF_SOURCE_INTERNAL;

2) внешний источник (ADC_VREF_SOURCE_EXTERNAL). В качестве источника используется напряжение, подаваемое на специальные выводы ADC0_REF+ и ADC1_REF−. При этом потребуется дополнительная схема, формирующая это напряжение.

При невысоких требованиях к точности и стабильности можно использовать внутренний источник опорного напряжения. Это гораздо проще, чем организовать внешний источник. Заметим, что встроенный датчик опорного напряжения VREF никакого отношения к источникам опорного напряжения АЦП не имеет. Его никак не удастся использовать для этих целей.

Внешний источник выбирают в случае, когда нужно добиться особо точных и стабильных результатов измерений. При этом надо дать указания разработчику схемы прибора, чтобы он предусмотрел внешний источник.

В нашем случае, конечно же, задействуем внутренний источник.

В поле ADC_IntVRefSource указывают вид источника для датчика опорного напряжения VREF = 1,23 В. Здесь есть два варианта: может быть выбран более точный источник с температурной компенсацией на основе встроенного температурного датчика (ADC_INT_VREF_SOURCE_EXACT) или менее точный без температурной компенсации (ADC_INT_VREF_SOURCE_INEXACT). Для нас сейчас это не важно, поэтому выберем вариант ADC_INT_VREF_SOURCE_INEXACT. Еще раз подчеркнем, что **датчик опорного напряжения никакого отношения к источнику опорного напряжения АЦП не имеет!**

Наконец, разрешим работу АЦП:

```
ADC1_Cmd (ENABLE);
```

С этого момента АЦП готов к работе.

3.7. Режим одиночного преобразования по одному каналу с опросом бита окончания преобразования

Процесс работы с АЦП в режиме одиночного преобразования по одному каналу с опросом бита окончания преобразования выглядит следующим образом:

1. Запуск преобразования.
2. Ожидание завершения преобразования с периодическим опросом бита окончания преобразования.
3. Считывание результатов преобразования.
4. Обработка результатов преобразования.
5. Пауза.
6. Возврат к пункту 1.

Рассмотрим этот процесс более подробно на примере задачи `U_ADC_Task_U_Function()`, реализованной в модуле `adc.c`. Эта задача производит периодическое измерение напряжения на выводе `PD6` и выдает результат на ЖКИ:

```
__task void U_ADC_Task_Function (void)
{
   uint32_t DU;    // Результат аналого-цифрового преобразования
   float U; // Измеренное напряжение

   while(1)
   {
      // Запустить процесс аналого-цифрового преобразования
      ADC1_Start();

      // Дождаться завершения преобразования, постоянно опрашивая
      // соответствующий флаг в регистре состояния АЦП1
      while ((ADC1_GetStatus() & ADC_STATUS_FLG_REG_EOCIF) == 0);
```

```
            // os_dly_wait (1);
            // Получить результат преобразования
            DU = ADC1_GetResult() & 0x0000FFFF;

            // Преобразование показаний АЦП в измеренное напряжение
            U = (U_ADC_U / U_ADC_D) * DU;

            // Вывести результат измерения напряжения на ЖКИ в вольтах
            sprintf(message , "U = %5.3fV", U);
            U_MLT_Put_String (message, 3);

            // Вывести результат аналого-цифрового преобразования
            sprintf(message , "ADC: 0x%1x", DU);
            U_MLT_Put_String (message, 4);

            os_dly_wait (250);
        }
    }
```

Запуск процесса преобразования производится путем вызова функции `ADC1_Start()`.

Выполнение программы на этом не приостанавливается: мы просто сообщаем микроконтроллеру о необходимости начать преобразование, занеся единицу в бит `ADC1_CFG_REG_GO` регистра управления АЦП `ADC1_CFG`.

В этом можно убедиться, просмотрев исходный код функции `ADC1_Start()`. Для этого достаточно подвести курсор мыши к имени функции `ADC1_Start()`, нажать правую кнопку мыши и выбрать пункт меню *Go to Definition of 'ADC1_Start'*.

Далее следует дождаться завершения преобразования, периодически опрашивая флаг завершения преобразования `ADC_STATUS_FLG_REG_EOCIF` в регистре состояния АЦП1:

```
while ((ADC1_GetStatus() & ADC_STATUS_FLG_REG_EOCIF) == 0);
```

В пустом цикле с помощью функции `ADC1_GetStatus()` мы периодически считываем значение бита `ADC_STATUS_FLG_REG_EOCIF` в регистре состояния АЦП1 (`ADC1_STATUS`). Пока идет преобразование, этот бит сохраняет значение ноль. Как только преобразование завершится, флаг `ADC_STATUS_FLG_REG_EOCIF` примет значение 1, и цикл будет завершен.

Такой способ ожидания прост, нагляден и надежен. Но недостаток его состоит в том, что процессор вынужден постоянно опрашивать флаг завершения, тратя лишнее время.

Если преобразования выполняются редко и время преобразования не имеет большого значения, можно рассмотреть еще более простой способ ожидания завершения преобразования:

```
ADC1_Start ();
os_dly_wait (1);
```

После запуска преобразования выполнение программы приостанавливается на время заведомо большее, чем потребное время преобразования. Например, на 1 мс. В течение этого времени процессор не будет выполнять данную задачу, переключившись на остальные.

Когда преобразование выполнено, необходимо получить его результаты. В простейшем случае это делается путем считывания значения специального регистра ADC1_RESULT. Проще всего это сделать, вызвав функцию ADC1_GetResult():

```
DU = ADC1_GetResult() & 0x0000FFFF;
```

В данном примере результат заносится в целочисленную переменную DU. Обратите внимание, что к результату вызова функции ADC1_GetResult() применяется маска 0x0000FFFF, сбрасывающая 16 старших битов. Это делается по той причине, что результат преобразования занимает лишь 12 младших битов. Старшие 16 битов возвращают вспомогательную информацию, которая в данном примере не требуется, и ее нужно очистить.

Полученные результаты преобразования подвергают дальнейшей обработке. В нашем случае нужно из показаний АЦП получить значение измеренного напряжения в вольтах. Это делается следующим образом. В заголовке adc.h задаются две калибровочные константы:

```
// Калибровка вольтметра путем указания полученного значения
// АЦП U_ADC_D для известного напряжения U_ADC_U на входе АЦП
#define U_ADC_U 3.190F
#define U_ADC_D 0xF90
```

В функции U_ADC_Task_U_Function() модуля adc.c производится пересчет показаний АЦП в напряжение на его входе:

```
uint32_t DU;        // Результат аналого-цифрового преобразования
float U;            // Измеренное напряжение
...
U = (U_ADC_U / U_ADC_D) * DU;
```

В константах U_ADC_U и U_ADC_D хранятся калибровочные данные вольтметра: U_ADC_U – напряжение, измеренное эталонным прибором; U_ADC_D – показание АЦП в момент измерения эталонным прибором. Эти данные получают экспериментально.

Переменную DU умножаем на отношение U_ADC_U к U_ADC_D и заносим результат в вещественную переменную U, в которой и будет содержаться искомое напряжение.

Далее значение переменной U выводится на ЖКИ, выдерживается пауза в 250 мс и вновь запускается преобразование. Таким образом, программа периодически (примерно 4 раза в секунду) производит измерение напряжения, поступающего с потенциометра.

Обратите внимание на то, как «дрожат» показания прибора. Это хорошо видно по младшему разряду в выводимых результатах измерений. Происходит это из-за различных помех, а также несовершенства источников опорного напряжения и питания АЦП. Стабильность показаний прибора можно значительно улучшить, применив усреднение результатов измерений, о чем будет рассказано далее.

3.8. Режим одиночного преобразования по одному каналу с прерыванием по окончанию преобразования

Теперь рассмотрим более совершенный способ работы с АЦП, при котором используется аппаратное прерывание. Но сначала несколько слов о самих прерываниях.

Аппаратным прерыванием (interrupt) называют сигнал от периферийного устройства, сообщающий процессорному ядру о наступлении определенного события. При этом выполнение текущей программы приостанавливается, и управление передается специальной функции обработки прерываний, которая реагирует на событие и обслуживает его, а затем возвращает

управление в прерванную программу. В микроконтроллерах семейства 1986ВЕ9х можно отслеживать прерывания от многих устройств, в том числе и от АЦП. При этом от некоторых устройств могут возникать прерывания по нескольким причинам.

Применительно к АЦП обычно используют прерывания по окончанию выполнения очередного аналого-цифрового преобразования, что и будет использовано нами.

Обработчики аппаратных прерываний применительно к микроконтроллерам семейства 1986ВЕ9х размещаются в модуле `MDR32F9Qx_it.c`. Для прерываний, которые не используются в проекте, функции-обработчики будут пустыми. Создавая обработчик прерывания, нужно стараться сделать его как можно короче и проще, чтобы излишне не загружать процессор.

В микроконтроллерах с архитектурой ARM32 используется контроллер вложенных векторных прерываний **NVIC** (Nested Vector Interrupt Controller). NVIC представляет собой специальную подсистему, управляющую аппаратными прерываниями. NVIC позволяет задать каждому прерыванию определенный приоритет, в соответствии с которым будет определяться порядок обработки запросов на прерывания, поступивших одновременно. Сначала обрабатывается запрос на прерывание с большим приоритетом. Остальные запросы ставятся в очередь типа FIFO (первый зашел – первый вышел).

Важно помнить, что любые настройки NVIC нужно производить до запуска ОСРВ RTX. После запуска RTX попытка настройки NVIC приведет зависанию микроконтроллера.

Процесс работы с АЦП при использовании прерывания по окончанию преобразования выглядит следующим образом:

1. Запуск преобразования.
2. **Ожидание сигнала из обработчика прерываний о завершении преобразования.**
3. Считывание результатов преобразования.
4. Обработка результатов преобразования.
5. Пауза.
6. Возврат к пункту 1.

По сравнению с предыдущим способом работы с АЦП все почти то же самое, за исключением пункта 2.

Откроем проект Lab3_2 и найдем исходный текст задачи U_ADC_Task_Function() в модуле adc.c.

```
__task void U_ADC_Task_Function (void)
{
   uint32_t DU;    // Результат аналого-цифрового преобразования
   float U;        // Измеренное напряжение

   while(1)
   {
      // Запустить процесс аналого-цифрового преобразования
      ADC1_Start();

      // Дождаться окончания преобразования
      os_evt_wait_or (EVENT_ADC_EOC, 0xFFFF);

      // Получить результат преобразования
      DU = ADC1_GetResult () & 0x0000FFFF;

      // Преобразование показаний АЦП в измеренное напряжение
      U = (U_ADC_U / U_ADC_D) * DU;

      // Вывести результат измерения напряжения на ЖКИ в вольтах
      sprintf(message , "U = %5.3fV", U);
      U_MLT_Put_String (message, 3);

      // Вывести результат аналого-цифрового преобразования
      sprintf(message , "ADC: 0x%1x", DU);
      U_MLT_Put_String (message, 4);

      os_dly_wait (250);
   }
}
```

По сравнению с предыдущим примером разница заключается в одной строке:

```
os_evt_wait_or (EVENT_ADC_EOC, 0xFFFF);
```

После запуска преобразования функцией задача ждет появления события EVENT_ADC_EOC, которое сигнализирует об окончании преобразования. Событие будет установлено в обработчике аппаратного прерывания от АЦП, который представляет собой функцию ADC_IRQHandler(), размещенную в специальном модуле MDR32F9Qx_it.c:

```
void ADC_IRQHandler (void)
{
   volatile uint32_t result;
   if (ADC1_GetITStatus (ADC1_IT_END_OF_CONVERSION))
   {
      result = ADC1_GetResult();
      isr_evt_set (EVENT_ADC_EOC, U_ADC_Task_ID);
   }
}
```

В обработчике проверяется из-за какого конкретно события, связанного с АЦП, возникло прерывание. Если из-за окончания преобразования ADC1, то считывается результат преобразования, чтобы сбросить флаг прерывания по завершению преобразования. Если этого не сделать, то микроконтроллер будет бесконечно заходить в подпрограмму-обработчик прерывания, т.е. зависнет. Затем устанавливается событие об окончании цикла аналого-цифрового преобразования с помощью функции `isr_evt_set()`. Задача `U_ADC_Task_Function()` получит это событие и продолжит свое выполнение, обработав результаты измерения.

Во время ожидания события процессорное время не тратится впустую. Микроконтроллер имеет возможность выполнять другие задачи, если таковые имеются. В этом и состоит огромное преимущество многозадачных операционных систем.

При инициализации АЦП в функцию `U_ADC_Init()` добавится несколько настроек.

Потребуется разрешить прерывания по окончанию аналого-цифрового преобразования:

```
ADC1_ITConfig (ADC1_IT_END_OF_CONVERSION, ENABLE);
```

Также нужно задать приоритет аппаратного прерывания от АЦП:

```
NVIC_SetPriority (ADC_IRQn, 1);
```

И, наконец, надо разрешить аппаратные прерывания от АЦП:

```
NVIC_EnableIRQ (ADC_IRQn);
```

3.9. Измерение температуры микроконтроллера с помощью АЦП

Внутри микроконтроллеров семейства 1986BE9x есть встроенный датчик температуры, который позволяет измерять температуру корпуса микроконтроллера. Датчик представляет собой специальную электронную схему, которая выдает аналоговое напряжение VTEMP, пропорциональное температуре кристалла микроконтроллера. Это напряжение подается на канал 31 АЦП (`ADC_CH_TEMP_SENSOR`) и далее может быть измерено с помощью АЦП (вернемся к рисунку 3.3).

Таким образом, у программиста имеется возможность следить за температурным режимом работы микроконтроллера. В примере `Lab3_3` данной работы (откройте этот проект) мы будем измерять температуру микроконтроллера и выводить ее на ЖКИ. Это реализовано в задаче `U_ADC_Task_Function()`.

Процесс измерения температуры практически аналогичен измерению напряжения с потенциометра, рассмотренному в примере `Lab3_2`. Отличие состоит лишь в разрешении работы датчика температуры, настройке канала АЦП (используется канал 31, а не 6) и обработке результатов измерений.

В функции `U_ADC_Init()` модуля `adc.c` при настройке общей конфигурации АЦП разрешим работу температурного датчика:

```
ADC_InitStructure.ADC_TempSensor = ADC_TEMP_SENSOR_Enable;
ADC_InitStructure.ADC_TempSensorAmplifier =
ADC_TEMP_SENSOR_AMPLIFIER_Enable;
ADC_InitStructure.ADC_TempSensorConversion =
ADC_TEMP_SENSOR_CONVERSION_Enable;
```

При выборе канала АЦП указываем канал для температурного датчика:

```
ADCx_InitStructure.ADC_ChannelNumber = ADC_CH_TEMP_SENSOR;
```

При обработке результатов требуется преобразовать значение `D`, полученное с АЦП в температуру, выраженную в градусах Цельсия. Для этого нужно иметь так называемые калибровочные данные, которые применительно к нашей задаче должны представлять собой результаты аналого-цифрового преобразования для двух разных температур, желательно сильно отличающихся. Например, для 25 ºС и 85 ºС.

В некоторых типах микроконтроллеров такие данные бывают занесены на заводе-изготовителе в специальные регистры индивидуально для каждого экземпляра микросхемы. К сожалению, для микроконтроллеров семейства 1986ВЕ9х таких данных нет. Поэтому при разработке серьезных устройств калибровку датчика температуры приходится производить самостоятельно, используя специальное оборудование (весьма дорогое!) под названием климатическая камера. В такую камеру помещают экземпляр устройства, создают требуемую температуру и снимают показания АЦП для температурного датчика.

Результаты такой калибровки представляют собой следующие четыре числа:

- $T1$ – температура в точке 1;
- $D1$ – значение АЦП в точке 1;
- $T2$ – температура в точке 2;
- $D2$ – значение АЦП в точке 2.

Искомая температура находится по формуле (3.1):

$$T = \frac{(D - D1) \cdot (T2 - T1)}{(D2 - D1)} + T1 \qquad (3.1)$$

Естественно, в рамках наших работ климатическую камеру мы использовать не будем, а возьмем приблизительные калибровочные данные.

В заголовке `adc.h` калибровочные данные представлены в виде следующих констант:

```
#define ADC_TS_T1 25.0F // Температура в 1-й точке
#define ADC_TS_D1 0x6A0 // Значение АЦП в 1-й точке
#define ADC_TS_T2 60.0F // Температура во 2-й точке
#define ADC_TS_D2 0x7A0 // Значение АЦП во 2-й точке
```

Программная реализация процесса вычисления температуры выглядит так:

```
T = ((int32_t)D - (int32_t) ADC_TS_D1) * (ADC_TS_T2 - ADC_TS_T1)
    / (ADC_TS_D2 - ADC_TS_D1) + ADC_TS_T1;
```

Значение вычисленной температуры далее выводится на ЖКИ.

3.10. Прямой доступ к памяти

3.10.1. Использование прямого доступа к памяти при работе с АЦП

Микроконтроллеры семейства 1986BE9x позволяют также работать с АЦП с использованием прямого доступа к памяти. **Прямой доступ к памяти** (англ. DMA – Direct Memory Access) представляет собой механизм, позволяющий периферийным устройствам напрямую, без использования центрального процессора, обращаться к оперативной памяти или памяти программ. Поскольку для сокращенного обозначения наибольшей популярностью пользуется английская аббревиатура **DMA**, то и мы также будем использовать ее.

Использование DMA позволяет существенно разгрузить центральный процессор, не отвлекая его на взаимодействие с периферийными устройствами. Кроме того, скорость работы периферийных устройств также существенно возрастает. Однако настройка DMA требует определенных знаний и является одним из наиболее сложных аспектов в работе с микроконтроллерами.

В рамках работы мы применим DMA для получения блока однотипных измерений с помощью АЦП. Как было видно ранее, результаты измерений напряжения требуют усреднения, чтобы получились стабильные показания. Для этого надо выполнить некоторое количество измерений и найти среднее арифметическое значение. Конечно, это можно сделать и без DMA, последовательно выполнив требуемое количество измерений, но гораздо эффективней это делается с использованием прямого доступа к памяти.

На рисунке 3.4 схематично показан процесс работы с АЦП без использования DMA, а на рисунке 3.5 – с использованием DMA. В первом случае процессор вынужден самостоятельно обращаться и к АЦП, и к ОЗУ. Во втором же случае взаимодействие между АЦП и ОЗУ ведется посредством DMA, а процессор в этом не участвует.

Рассмотрим процесс использования DMA. Для этого откроем проект `Lab3_4`, с помощью которого выполняется измерение температуры корпуса микроконтроллера с использованием встроенного датчика температуры. Заглянем в модуль `adc.c`.

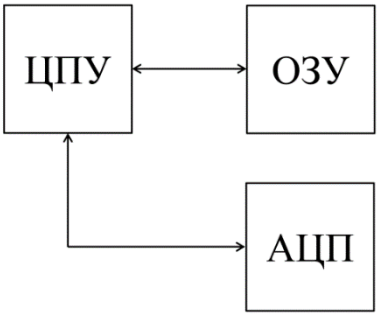


Рисунок 3.4 – Работа с АЦП без DMA

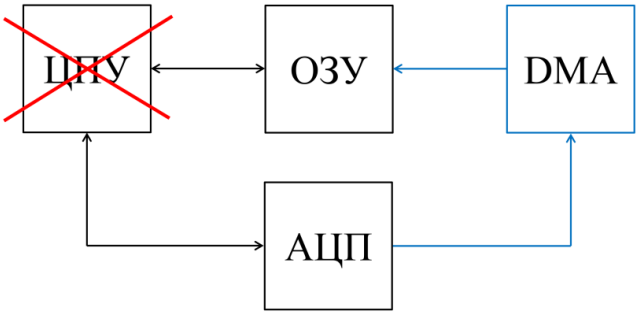


Рисунок 3.5 – Работа с АЦП с использованием DMA

Измерение температуры корпуса микроконтроллера с использованием DMA производится в задаче `U_ADC_Task_Function()`:

```
__task void U_ADC_Task_Function (void)
{
   uint32_t D;  // Результат аналого-цифрового преобразования
   float T;     // Измеренная температура
   uint32_t i;

   while(1)
   {
      // Разрешить работу DMA с АЦП, запускается цикл
      // аналого-цифровых преобразований
      DMA_Cmd (DMA_Channel_ADC1, ENABLE);
```

```
            // Дождаться окончания преобразования
            os_evt_wait_or (EVENT_ADC_EOC, 0xFFFF);

            // Усреднить результат
            for (i = 0, D = 0; i < U_ADC_BUFFER_SIZE; i++)
            D += ADC_Buffer[i];
            D /= U_ADC_BUFFER_SIZE;

            // Преобразование показаний АЦП в температуру в градусах Цельсия
            T = ((int32_t)D - (int32_t) ADC_TS_D1) * (ADC_TS_T2 - ADC_TS_T1)
                / (ADC_TS_D2 - ADC_TS_D1) + ADC_TS_T1;

            // Вывести результат измерения температуры на ЖКИ
            sprint (message , "\xD2\xE5\xEC\xEF\xE5\xF0.:%6.1f\xB0\x43", T);
            U_MLT_Put_String (message, 3);

            // Вывести результат аналого-цифрового преобразования
            sprint (message , "ADC: 0x%1x", D);
            U_MLT_Put_String (message, 4);

            // Пауза в тиках системного таймера. Здесь 1 тик = 1 мс
            os_dly_wait (250);
        }
    }
```

Процесс преобразования запускается путем вызова функции `DMA_Cmd()`, разрешающей работу DMA совместно с АЦП. После этого задача ждет появления события `EVENT_ADC_EOC`, которое сигнализирует об окончании цикла преобразования:

```
os_evt_wait_or(EVENT_ADC_EOC, 0xFFFF);
```

Событие будет установлено в обработчике аппаратного прерывания от DMA, который представляет собой функцию, размещенную в специальном модуле `MDR32F9Qx_it.c`:

```
// Обработчик для прерывания от DMA
void DMA_IRQHandler (void)
{
   // Подготовить к работе новый цикл аналого-цифровых преобразований
   DMA_InitStructure.DMA_CycleSize = U_ADC_BUFFER_SIZE;
   DMA_Init (DMA_Channel_ADC1, &DMA_Channel_InitStructure);
```

```
    // Запретить дальнейшую работу канала DMA с ADC
    DMA_Cmd (DMA_Channel_ADC1, DISABLE);

    // Установить событие об окончании цикла
    // аналого-цифрового преобразования
    isr_evt_set (EVENT_ADC_EOC, U_ADC_Task_ID);
}
```

АЦП последовательно выполнит количество преобразований, определяемое константой `U_ADC_BUFFER_SIZE`, описанной в заголовке `adc.h` (в нашем случае 256 преобразований). Результаты будут помещены в массив `ADC_Buffer`. Заметим, что процессор совершенно не будет отвлекаться на эту работу, а будет заниматься выполнением других задач.

Как только последнее из 256 преобразований будет выполнено, DMA сгенерирует аппаратное прерывание, и процессор переключится на выполнение функции-обработчика аппаратных прерываний от канала DMA (`DMA_Channel_ADC1`).

Первым делом при этом необходимо подготовить к работе новый цикл аналого-цифровых преобразований и запретить дальнейшую работу канала DMA с АЦП. Если этого не сделать, процессор будет бесконечно заходить в эту функцию-обработчик прерывания и, очевидно, зависнет.

Затем необходимо сообщить задаче `U_ADC_Task_Function`, ожидающей результатов преобразований, о том, что они готовы. Это делается путем установки сообщения `EVENT_ADC_EOC`:

```
isr_evt_set (EVENT_ADC_EOC, U_ADC_Task_ID);
```

После этого задача `U_ADC_Task_Function` продолжит свое выполнение, усреднив результат преобразований:

```
for (i = 0, D = 0; i < U_ADC_BUFFER_SIZE; i++)
   D += ADC_Buffer[i];
   D /= U_ADC_BUFFER_SIZE
```

Далее усредненный результат преобразуется к температуре уже рассмотренным ранее способом:

```
T = ((int32_t) D - (int32_t) ADC_TS_D1) * (ADC_TS_T2 - ADC_TS_T1)
    / (ADC_TS_D2 - ADC_TS_D1) + ADC_TS_T1;
```

3.10.2. Настройка прямого доступа к памяти для работы с АЦП

Теперь рассмотрим процесс настройки DMA для совместной работы с АЦП.

Совокупность настроек DMA в целом располагаем в структуре DMA_InitStructure:

```
DMA_CtrlDataInitTypeDef DMA_InitStructure;
```

Настройки канала DMA располагаем в структуре DMA_Channel_InitStructure:

```
DMA_ChannelInitTypeDef DMA_Channel_InitStructure;
```

Заметим, что обе структуры являются глобальными, поскольку с ними придется работать и за пределами модуля generator.c, а именно – в модуле обработчиков аппаратных прерываний MDR32F9Qx_it.c.

В модуле adc.c настройки DMA сосредоточены в функции DMA_Config():

```
static void DMA_Config(void)
{
  // Разрешить тактирование DMA
  RST_CLK_PCLKcmd (RST_CLK_PCLK_DMA | RST_CLK_PCLK_SSP1 |
                   RST_CLK_PCLK_SSP2, ENABLE);
  // Запретить все прерывания, в том числе от SSP1 и SSP2
  NVIC->ICPR[0] = 0xFFFFFFFF;
  NVIC->ICER[0] = 0xFFFFFFFF;

  // Сбросить все настройки DMA
  DMA_DeInit();
  DMA_StructInit (&DMA_Channel_InitStructure);
  DMA_InitStructure.DMA_SourceBaseAddr =
                              (uint32_t)(&(MDR_ADC->ADC1_RESULT));
  DMA_InitStructure.DMA_DestBaseAddr = (uint32_t) &ADC_Buffer;
  DMA_InitStructure.DMA_CycleSize = U_ADC_BUFFER_SIZE;
  DMA_InitStructure.DMA_SourceIncSize = DMA_SourceIncNo;
  DMA_InitStructure.DMA_DestIncSize = DMA_DestIncHalfword;
  DMA_InitStructure.DMA_MemoryDataSize =
                                      DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_NumContinuous = DMA_Transfers_1;
  DMA_InitStructure.DMA_SourceProtCtrl = DMA_SourcePrivileged;
  DMA_InitStructure.DMA_DestProtCtrl = DMA_DestPrivileged;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Basic;
```

```
    // Задать структуру канала
    DMA_Channel_InitStructure.DMA_PriCtrlData = &DMA_InitStructure;
    DMA_Channel_InitStructure.DMA_Priority = DMA_Priority_Default;
    DMA_Channel_InitStructure.DMA_UseBurst = DMA_BurstClear;
    DMA_Channel_InitStructure.DMA_SelectDataStructure =
                                                DMA_CTRL_DATA_PRIMARY;

    // Инициализировать канал
    DMA DMA_Init (DMA_Channel_ADC1, &DMA_Channel_InitStructure);

    MDR_DMA->CHNL_REQ_MASK_CLR = 1 << DMA_Channel_ADC1;
    MDR_DMA->CHNL_USEBURST_CLR = 1 << DMA_Channel_ADC1;

    // Разрешить работу DMA с каналом АЦП1
    DMA_Cmd (DMA_Channel_ADC1, ENABLE);

    // Задать приоритет аппаратного прерывания от DMA
    NVIC_SetPriority (DMA_IRQn, 1);
}
```

Вначале нужно разрешить тактирование DMA и обязательно интерфейсов SSP1 и SSP2:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_DMA | RST_CLK_PCLK_SSP1 |
                 RST_CLK_PCLK_SSP2, ENABLE);
```

По логике вещей, последовательные интерфейсы SSP1 и SSP2 не имеют непосредственного отношения к DMA, хотя и могут работать с ним. Но в микроконтроллеры семейства 1986BE9x вкралась аппаратная ошибка, в результате которой невозможно корректно работать с DMA, если SSP1 и SSP2 не тактируются. Поэтому возьмем за правило включать тактирование интерфейсов SSP1 и SSP2, когда DMA задействован в проекте. Если этого не сделать, программа попросту зависнет после возникновения прерывания от DMA.

По той же причине на время настройки DMA нужно отключить все аппаратные прерывания:

```
NVIC->ICPR[0] = 0xFFFFFFFF;
NVIC->ICER[0] = 0xFFFFFFFF;
```

Деинициализируем, т.е. сбросим настройки DMA:

```
DMA_DeInit();
```

Инициализируем структуру для настройки DMA, передав в качестве параметра адрес структуры для настройки требуемого канала DMA:

```
DMA_StructInit (&DMA_Channel_InitStructure);
```

Далее заполняем структуру для настройки DMA.

Зададим адрес источника данных, в качестве которого будет использоваться регистр результатов преобразования данных АЦП1 – ADC1_RESULT. Оттуда будут браться данные с помощью DMA.

```
DMA_InitStructure.DMA_SourceBaseAddr =
                                (uint32_t)(&(MDR_ADC->ADC1_RESULT));
```

Укажем адрес приемника данных. В качестве него используется буфер (массив) результатов измерений ADC_Buffer. В его элементы будут записываться результаты аналого-цифрового преобразования:

```
DMA_InitStructure.DMA_DestBaseAddr = (uint32_t) &ADC_Buffer;
```

Зададим размер буфера результатов измерений, т.е. определим, сколько раз мы произведем измерения:

```
DMA_InitStructure.DMA_CycleSize = U_ADC_BUFFER_SIZE;
```

Запретим автоматическое увеличение адреса источника, поскольку читаем данные всегда из одного и того же регистра ADC1_RESULT:

```
DMA_InitStructure.DMA_SourceIncSize = DMA_SourceIncNo;
```

Разрешаем автоматическое увеличение адреса приемника на полслова (т.е. на 2), поскольку записываем данные в разные элементы буфера результатов измерений ADC_Buffer:

```
DMA_InitStructure.DMA_DestIncSize = DMA_DestIncHalfword;
```

Указываем, по сколько байт будем передавать за один раз. В нашем случае по два байта или, по-другому, по полслова:

```
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
```

Казалось бы, зачем нужна эта настройка? Ведь мы только что указали, что адрес приемника будет автоматически увеличиваться на те же два байта. Однако это разные вещи. Элементы массива могут иметь больший размер, чем количество данных, передаваемых за один раз. Например, мы можем использовать для хранения результатов измерений массив с элементами по 4 байта, но передавать за один раз лишь два байта. При этом у нас будут следующие настройки:

```
DMA_InitStructure.DMA_DestIncSize = DMA_DestIncWord;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
```

Таким образом, адрес автоматически увеличивается на 4 байта, а передаем за раз лишь два байта.

Далее укажем, сколько раз будем повторять полные циклы передачи данных:

```
DMA_InitStructure.DMA_NumContinuous = DMA_Transfers_1;
```

В нашем случае требуется выполнить цикл лишь один раз. Это означает, что DMA один раз выполнит передачу `U_ADC_BUFFER_SIZE` элементов данных, а затем остановится. Можно выбирать значения: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 и 1024. Режим бесконечного повтора передачи, к сожалению, не предусмотрен.

Включим режимы защиты источника и приемника:

```
DMA_InitStructure.DMA_SourceProtCtrl = DMA_SourcePrivileged;
DMA_InitStructure.DMA_DestProtCtrl = DMA_DestPrivileged;
```

Смысл этих двух настроек не будем сейчас рассматривать.

Также зададим так называемый базовый режим работы DMA:

```
DMA_InitStructure.DMA_Mode = DMA_Mode_Basic;
```

Помимо базового, есть еще целый ряд других режимов работы DMA. Но объяснение их смысла займет много времени и отвлечет нас от основной цели данной работы. Тем, кто заинтересовался этим вопросом, можно порекомендовать обратиться к фирменной документации на микроконтроллеры семейства

1986ВЕ9х. Там этой теме посвящен не один десяток страниц. Пока лишь запомним, что совместно с АЦП чаще используется базовый режим работы DMA.

Отметим также, что при работе DMA совместно с АЦП нередко используется режим «пинг-понг» (DMA_Mode_PingPong), при котором можно поочередно класть результаты измерения в два разных массива. Это позволяет за время обработки результатов измерений, сохраненных в первом массиве, выполнять новый цикл измерений, складывая результаты во второй массив. Такой подход полезен, если необходимо непрерывно производить какие-нибудь измерения.

Теперь заполним структуру для настройки канала DMA.

К первичному блоку настроек канала привяжем ранее заполненную структуру настройки DMA и выберем ее в качестве используемой в данный момент:

```
DMA_Channel_InitStructure.DMA_PriCtrlData = &DMA_InitStructure;
DMA_Channel_InitStructure.DMA_SelectDataStructure =
                                                DMA_CTRL_DATA_PRIMARY;
```

Вообще-то есть два блока настроек канала DMA: первичный (DMA_PriCtrlData) и альтернативный (DMA_AltCtrlData). Альтернативный требуется лишь при режиме пинг-понг, поэтому нами сейчас не рассматривается.

Приоритет канала DMA установим по умолчанию:

```
DMA_Channel_InitStructure.DMA_Priority = DMA_Priority_Default;
```

Снимем запрет на одиночные запросы к DMA:

```
DMA_Channel_InitStructure.DMA_UseBurst = DMA_BurstClear;
```

Полученной структурой инициализируем канал DMA для работы с таймером АЦП1:

```
DMA_Init (DMA_Channel_ADC1, &DMA_Channel_InitStructure);
```

Снимем запрет на запросы к DMA со стороны АЦП1. Это рекомендуется для нормальной работы DMA (что-то вроде заплатки для устранения аппаратных ошибок в микроконтроллерах 1986ВЕ9х):

```
MDR_DMA->CHNL_REQ_MASK_CLR = 1 << DMA_Channel_ADC1;
MDR_DMA->CHNL_USEBURST_CLR = 1 << DMA_Channel_ADC1;
```

Разрешим работу DMA с каналом для АЦП1:

```
DMA_Cmd (DMA_Channel_ADC1, ENABLE);
```

Зададим приоритет 1 для аппаратного прерывания от DMA:

```
NVIC_SetPriority (DMA_IRQn, 1);
```

В функции `ADC_Config()` производится настройка АЦП. Она практически аналогична тому, как это сделано в примере `Lab3_3`. Разница совсем невелика.

Вместо режима одиночного преобразования выбирается режим циклического (многократного) преобразования:

```
ADCx_InitStructure.ADC_SamplingMode = ADC_SAMPLING_MODE_CICLIC_CONV;
```

Вместо прерываний от ADC разрешаются прерывания от DMA:

```
NVIC_EnableIRQ (DMA_IRQn);
```

3.11. Режим многократного преобразования с автоматическим переключением нескольких каналов и использованием прямого доступа к памяти

Наконец, рассмотрим весьма интересный и полезный режим работы АЦП с автоматическим переключением нескольких каналов. Смысл этого режима состоит в следующем. При настройке АЦП программист указывает несколько каналов, по которым будут производиться преобразования. Например, выбраны каналы 2, 5 и 7. При первом запуске преобразования будет автоматически выбран канал 2. При втором – канал 5, при третьем – 7, при четвертом – опять канал 2 и т.д.

Такой подход полезен, если нужно организовать регулярное преобразование сразу нескольких сигналов. Например, если к контроллеру подключено 4 внешних датчика температуры. Можно, конечно, переключать каналы вручную, но на это потребуется много процессорного времени.

Автоматическое переключение каналов особенно эффективно при использовании совместно с DMA. Такой подход рассмотрен в примере Lab3_5. С помощью АЦП измеряется напряжение, снимаемое с внешнего потенциометра, подключенного к каналу ADC6, и температура кристалла микроконтроллера.

Настройки АЦП реализованы в функции ADC_Config() модуля adc.c. Они очень похожи на предыдущий пример Lab3_4. Приведем лишь отличающиеся строки:

```
ADCx_InitStructure.ADC_ChannelSwitching =
ADC_CH_SWITCHING_Enable;
ADCx_InitStructure.ADC_ChannelNumber = 0;
ADCx_InitStructure.ADC_Channels = U_ADC_U_CH_MSK |
                                  ADC_CH_TEMP_SENSOR_MSK;
```

Разрешается автоматическое переключение каналов, номер канала для постоянной работы не выбирается (0). Задаются два канала (U_ADC_U_CH_MSK – он же ADC6 и ADC_CH_TEMP_SENSOR_MSK – канал температурного датчика), между которыми будет производиться автоматическое переключение.

Настройка DMA совершенно аналогична предыдущему примеру.

Измерения требуемых параметров производятся с помощью задачи U_ADC_Task_U_T_DMA_Function(), представленной в модуле adc.c:

```
__task void U_ADC_Task_U_T_DMA_Function (void)
{
   uint32_t DU, DT;      // Результаты АЦП
   uint32_t KU, KT;      // Количество результатов преобразований
   float T;        // Измеренная температура
   float U;        // Измеренное напряжение
   uint32_t i;

   while(1)
   {
      // Разрешить работу АЦП1
      // Запускается цикл преобразований
      ADC1_Cmd (ENABLE);

      // Дождаться окончания преобразования
      os_evt_wait_or (EVENT_ADC_EOC, 0xFFFF);
```

```
            // Усреднить результаты преобразования
            for (i = 0, DU = 0, DT = 0, KU = 0, KT = 0;
            i < U_ADC_BUFFER_SIZE; i++)
            {
               // По какому каналу АЦП было выполнено преобразование?
               switch(ADC_Buffer[i].Word[1])
               {
                  // Канал измерения напряжения
                  case U_ADC_U_CH:
                  DU += ADC_Buffer[i].Word[0];
                  KU++;
                  break;

                  // Канал измерения температуры
                  case ADC_CH_TEMP_SENSOR:
                  DT += ADC_Buffer[i].Word[0];
                  KT++;
                  break;
               }
            }

         // Если были преобразования для канала напряжения
         if(KU)
         DU /= KU;

         // Если были преобразования для канала температуры
         if(KT)
            DT /= KT;

         // Преобразование показаний АЦП в температуру
         T = ((int32_t)DT - (int32_t) ADC_TS_D1) * (ADC_TS_T2 - ADC_TS_T1)
              / (ADC_TS_D2 - ADC_TS_D1) + ADC_TS_T1;

         // Преобразование показаний АЦП в измеренное напряжение
         U = (U_ADC_U / U_ADC_D) * DU;

         // Вывести результат измерения напряжения на ЖКИ
         sprintf(message , "U = %5.3fV", U);
         U_MLT_Put_String (message, 3);

         // Вывести результат измерения температуры на ЖКИ
         sprintf(message , "\xD2\xE5\xEC\xEF\xE5\xF0.:%6.1f\xB0\x43", T);
         U_MLT_Put_String (message, 5);

         os_dly_wait (250);
         }
}
```

Запуск серии аналого-цифровых преобразований и ожидание их окончания реализованы точно так же, как и в примере Lab3_4. Обработчик прерываний от DMA тоже ничем не отличается от предыдущего примера, поэтому его исходный текст здесь не приводится. Интерес представляет процесс обработки полученных данных, а также способ организации буфера результатов преобразований.

Все результаты преобразований заносятся в один буфер ADC_Buffer, по-другому сделать нельзя, так как в настройках DMA мы можем указать адрес лишь одного буфера. Следовательно, нужно иметь возможность различать результаты преобразования для каждого из каналов. Здесь нельзя полагаться на кажущийся естественный порядок следования каналов, заданных в настройках АЦП:

```
ADCx_InitStructure.ADC_Channels = U_ADC_U_CH_MSK |
                                  ADC_CH_TEMP_SENSOR_MSK;
```

Нет никакой гарантии, что микроконтроллер обязательно начнет преобразования с канала U_ADC_U_CH_MSK или, наоборот, с канала ADC_CH_TEMP_SENSOR_MSK. Следовательно, нельзя считать, что четные элементы буфера всегда будут относиться к такому-то каналу, а нечетные – к такому-то.

Но решение проблемы есть. Дело в том, что вместе с результатами аналого-цифрового преобразования в регистр ADC1_RESULT заносится также и номер канала, по которому производилось преобразование.

Бит 0…11 в 32-разрядном регистре ADC1_RESULT содержат результат преобразования, а биты 16…20 – номер канала. Следовательно, нужно сохранять результаты преобразования в массиве из 32-разрядных слов.

Все вроде бы просто, но как лучше добраться до отдельных битов в таком слове? Наверное, наиболее рациональным способом является использование так называемых объединений. Посмотрите, как определен массив ADC_Buffer в модуле adc.c:

```
static TVariant32 ADC_Buffer [U_ADC_BUFFER_SIZE];
```

Необычный тип TVariant32 определен в заголовке variant.h:

```
// Объединение для работы с четырехбайтным числом
typedef volatile union
{
   uint32_t DWord; // Доступ к 4-байтному слову
   float Float;    // Доступ к вещественному значению (4 байта)
   uint8_t Byte[4];      // Доступ к отдельным байтам
   uint16_t Word[2];     // Доступ к отдельным словам
} TVariant32;
```

Возможно, читатель уже хорошо знаком с объединениями, но наблюдения показывают, что далеко не все программисты умеют ими пользоваться. Поэтому приведем немного теории.

Объединение (англ. union) – это особый тип данных в языке Си, который представляет собой несколько поименованных полей, физически занимающих одни и те же ячейки памяти. Вот как можно работать с объединениями.

Опишем следующие переменные:

```
TVariant32 x;
uint32_t a;
uint16_t b;
uint8_t c;
```

Присвоим полю `DWord` переменной `x` значение `0xAABBCCDD`:

```
x.DWord = 0xAABBCCDD
```

Теперь выполним следующие присваивания, чтобы понять, какие данные будут доступны через те или иные поля объединения:

```
a = x.DWord;      // 0xAABBCCDD
b = x.Word[0];    // 0xCCDD
b = x.Word[1];    // 0xAABB
c = x.Byte[0];    // 0xDD
c = x.Byte[1];    // 0xCC
c = x.Byte[2];    // 0xBB
c = x.Byte[3];    // 0xAA
```

Таким образом, мы имеем возможность удобно обращаться к отдельным байтам и полусловам 32-разрядного слова, что нам и требуется в примере `Lab3_5`. Посмотрите, как это делается в нашем примере:

```
// Усреднить результаты преобразования
for (i = 0, DU = 0, DT = 0, KU = 0, KT = 0;
     i < U_ADC_BUFFER_SIZE; i++)
{
   // По какому каналу АЦП было выполнено преобразование?
   switch(ADC_Buffer[i].Word[1])
   {
      // Канал измерения напряжения
      case U_ADC_U_CH:
      DU += ADC_Buffer[i].Word[0];
      KU++;
      break;

      // Канал измерения температуры
      case ADC_CH_TEMP_SENSOR:
      DT += ADC_Buffer[i].Word[0];
      KT++;
      break;
   }
}
```

С помощью поля `ADC_Buffer[i].Word[1]` мы определяем канал, по которому было сделано преобразование, обратившись, таким образом, к битам 16..20. А при помощи поля `ADC_Buffer[i].Word[0]` мы берем данные из разрядов 0…11, получив результат аналого-цифрового преобразования.

Типы, подобные `TVariant32`, часто называют вариантными, они в том или ином виде реализованы во многих современных языках. Заголовок `variant.h` и тип `TVariant32` не являются стандартными, но автор часто использует его в своих проектах.

После суммирования результатов преобразований нужно поделить полученные суммы на количество этих преобразований. Чтобы программист мог спать спокойно, лучше гарантированно исключить деление на ноль:

```
// Если были преобразования для канала напряжения
if(KU)
   DU /= KU;

// Если были преобразования для канала температуры
if(KT)
   DT /= KT;
```

На основе полученных результатов уже рассмотренным в предыдущих примерах способом вычисляются и выводятся на ЖКИ значения напряжения и температуры.

В заключение заметим, что количество элементов буфера ADC_Buffer должно быть удвоенным. Так, если для каждого канала требуется выполнить серию из 32 преобразований, размер буфера в заголовке adc.h будет описан так:

```
#define U_ADC_BUFFER_SIZE 2*32
```

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 3.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. Для проекта `Lab3_1` выполните калибровку измерителя напряжения, сравнив показания на ЖКИ с показаниями мультиметра и подобрав значения калибровочных констант. Добейтесь, чтобы значение индикации на отладочной плате совпадало со значением на мультиметре.

Пояснение. Калибровочные константы именуются `U_ADC_U` и `U_ADC_D` и содержатся в модуле `adc.h`.

2. Убедитесь, что в проекте `Lab3_3` производится измерение температуры в градусах Цельсия (ºC). Для этого прислоните палец (без лишних усилий!) к корпусу микроконтроллера и подержите примерно 20 секунд. Температура немного увеличится.

Измените программу так, чтобы температура также выводилась в Кельвинах (К) и градусах Фаренгейта (ºF).

3. На основе проекта `Lab3_4` реализуйте вольтметр (как в проекте `Lab3_1`), но с использованием DMA при работе с АЦП. Убедитесь, что показания на ЖКИ стали более стабильными, чем в проекте `Lab3_1`.

4. На основе проекта `Lab3_5` организуйте измерения сразу по трем каналам: напряжение с внешнего потенциометра, температура со встроенного датчика и напряжение со встроенного датчика VREF. Напряжения выводите в мВ, температуру – в градусах Цельсия.

Пояснение. Канал напряжения встроенного датчика именуется `ADC_CH_INT_VREF`.

Контрольные вопросы

1. Как выполняется измерение напряжения с помощью мультиметра М-838?
2. Что такое АЦП? Для чего он нужен?
3. Какие основные характеристики АЦП вы знаете?
4. Сколько уровней напряжения может различать 12-разрядный АЦП?
5. В каком диапазоне может измерять напряжение АЦП?
6. Какие основные режимы работы АЦП вы знаете?
7. Что такое DMA? Для чего его используют?

8. Как можно сообщить задаче об окончании аналого-цифрового преобразования?

9. Что такое обработчик прерывания?

10. К какому каналу АЦП подключен датчик температуры, встроенный в данный микроконтроллер?

11. Каким образом можно привести показания АЦП от температурного датчика к значению температуры в градусах Цельсия?

12. В чем состоит смысл режима работы АЦП с автоматическим переключением каналов?

Глава 4

Цифро-аналоговый преобразователь

Цель работы: получение навыков работы с цифро-аналоговым преобразователем, прямым доступом к памяти и цифровым осциллографом.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- мультиметр М-838 или его аналог;
- цифровой осциллограф-приставка Oscill;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- драйвер для осциллографа-приставки Oscill;
- приложение Oscill для осциллографа-приставки Oscill;
- примеры кода программ.

4.1. Подготовка к работе

Отключите питание отладочной платы, если оно было подключено, и переставьте перемычку «DAC_OUT_SEL» на плате в положение «EXT_CON». Это позволит соединить вывод PE0 с байонетным разъемом «DAC_OUT».

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab4_1\MDK-ARM`. Подключите к отладочной плате программатор и питание (подробности в разделе 1.5). Проверьте правильность настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

4.2. Описание проектов

Пример `Lab4_1` позволяет однократно выполнить цифро-аналоговое преобразование, выдав на выводе PE0 постоянное напряжение заданного уровня.

В примере `Lab4_2` с помощью ЦАП выполняется генерирование аналогового сигнала заданной формы: синусоидального или пилообразного. Сигнал получают на выводе PE0 микроконтроллера.

4.3. Понятие цифро-аналогового преобразователя

Цифро-аналоговый преобразователь (ЦАП или, по-английски, DAC – Digital to Analog Converter) представляет собой устройство, преобразующее цифровой код в пропорциональный уровень напряжения аналогового сигнала. Т.е. на выводе микроконтроллера, к которому подведен ЦАП, можно получить произвольный (с некоторым шагом) уровень напряжения. ЦАП применяют, например, в синтезаторах звука (звуковых картах), формирователях сигналов сложной формы и др. Основными характеристиками ЦАП являются разрядность и время преобразования.

В микросхемах семейства 1986BE9x имеется два ЦАП, позволяющих одновременно производить два независимых преобразования. Будем условно называть их DAC1 и DAC2. Каждый ЦАП имеет разрядность 12 бит. Это позволяет получать на выходе ЦАП до 2^{12} = 4096 различных уровней напряжения.

Однако в микросхеме K1986BE92QI, с которой мы работаем, доступен лишь один ЦАП – DAC2.

Так же, как и АЦП, ЦАП необходимо обеспечить напряжением питания AUCC, которое подводится к ножке 20 микросхемы K1986BE92QI. Этим же напряжением питаются и остальные аналоговые части микроконтроллера. Также имеется и дополнительный вывод 19 – аналоговая земля AGND.

В простейшем случае питание цифровой и аналоговой части микроконтроллера объединяют, но при этом точность цифро-аналоговых и аналого-цифровых преобразований существенно снижается. Именно так сделано на используемой нами отладочной плате. Если требуется повышенная точность преобразований, то питание аналоговой части организуют от отдельного источника напряжения.

Кроме того, для функционирования ЦАП необходим источник опорного напряжения UREF (опора ЦАП, как часто говорят). В качестве него можно использовать напряжение питания аналоговой части AUCC или подключать отдельный внешний источник опорного напряжения. Переключение между источниками опорного напряжения производится программно.

В нашем случае в качестве опоры используется AUCC. То есть опорное напряжение для ЦАП будет таким же, каким питаем аналоговую часть. У нас эти напряжения будут равны: UREF = AUCC = UCC = 3,3 В.

Отдельный источник опорного напряжения применяют в особо ответственных случаях, когда требуется высокая стабильность напряжения от температуры окружающей среды и во времени.

Входными данными для работы ЦАП является цифровой код, который заносят в специальные регистры DAC1_DATA или DAC2_DATA (в случае микроконтроллера K1986BE92QI – лишь DAC2_DATA). Выходным сигналом является напряжение DACOUT, формируемое на соответствующей ножке микросхемы – выходе ЦАП. Такой ножкой в микроконтроллере K1986BE92QI является ножка 18 (DAC_OUT2 – выход второго ЦАП). Заметим, что DAC_OUT2 находится на той же ножке, что и линия ввода-вывода PE0.

Выходное напряжение ЦАП, работающего в режиме с разрядностью 12 бит, определяется по очень простой формуле (4.1):

$$\mathrm{DACOUT} = \mathrm{UREF}\ \frac{\mathrm{DAC2\_DATA}}{4095} \qquad (4.1)$$

Число 4095 в знаменателе – это число на единицу меньшее, чем 4096. ☺ А 4096 – число возможных уровней выходного напряжения с учетом разрядности ЦАП в 12 бит, о чем писалось выше. Чтобы не запутаться с числами 4095 и 4096, надо вспомнить, что коду 0 соответствует напряжение 0 В. Это тоже одно из возможных уровней выходного напряжения.

Таким образом, по формуле (4.1) получаем линейное преобразование цифрового кода ЦАП в значение выходного аналогового напряжения. Конечно, количество уровней выходного напряжения не бесконечно, но весьма велико. В результате формируемый ЦАП выходной сигнал будет

состоять как бы из ступенек с маленьким шагом DACSTEP, определяемым формулой (4.2):

$$\text{DACSTEP} = \frac{\text{UREF}}{4095} \tag{4.2}$$

В нашем случае, как уже говорилось, опорное напряжение UREF = 3,3 В. Следовательно:

$$\text{DACSTEP} = \frac{3{,}3\ \text{В}}{4095} \approx 8{,}059 \cdot 10^{-4}\ \text{В} = 0{,}8059\ \text{мВ} \tag{4.3}$$

Шаг получается весьма небольшой. Вспомним, к примеру, что у обычной пальчиковой батареи напряжение 1,5 вольта, а тут – 0,8 **милли**вольта, т.е. почти в 2000 раз меньше.

Таким образом, с помощью ЦАП мы имеем возможность получить на выводе микроконтроллера требуемое напряжение из диапазона 0…3,300 В.

4.4. Настройка цифро-аналогового преобразователя

Перед началом использования, ЦАП, естественно, необходимо инициализировать. Рассмотрим, как это делается на примере проекта `Lab4_1`. Откроем модуль `dac.c` и рассмотрим функцию `U_DAC_Init`.

Выход DAC2 подведен к выводу микросхемы PE0. Поэтому необходимо соответствующим образом конфигурировать это вывод. Для этого, во-первых, нужно разрешить тактирование порта PORTE, а заодно и самого ЦАП:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_DAC | RST_CLK_PCLK_PORTE, ENABLE);
```

Во-вторых, нужно сделать вывод PE0 аналоговым. Для этого используем структуру `PortInitStructure`:

```
PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_Pin = PORT_Pin_0;
PortInitStructure.PORT_OE = PORT_OE_OUT;
PortInitStructure.PORT_MODE = PORT_MODE_ANALOG;
PORT_Init (MDR_PORTE, &PortInitStructure);
```

Далее, на всякий случай, сбросим все настройки ЦАП и произведем инициализацию DAC2:

```
DAC_DeInit();
DAC2_Init(DAC2_AVCC);
```

При этом в функции DAC2_Init в качестве параметра указывается тип источника опорного напряжения:

- DAC2_AVCC – в качестве источника опорного напряжения используется AUCC;
- DAC2_REF – используется внешний источник опорного напряжения, подключенный к выводу PE1.

Следующим этапом разрешаем работу DAC2:

```
DAC2_Cmd (ENABLE);
```

Как только мы выполним эту команду, вывод PE0 автоматически соединяется с выходом ЦАП. Теперь можно приступать к работе с ЦАП.

4.5. Работа с цифро-аналоговым преобразователем

Запуск процесса одиночного преобразования производится путем вызова функций DAC1_SetData или DAC2_SetData. В нашем случае используется DAC2, поэтому применяем функцию DAC2_SetData. В следующем примере показывается, как запустить DAC2, подав на его вход цифровое значение 1300:

```
DAC2_SetData (1300);
```

Выполнение программы на этом не приостанавливается: мы просто сообщаем микроконтроллеру о необходимости начать преобразование.

Интересно отметить, что в микроконтроллерах семейства 1986BE9x не предусмотрено штатных средств для отслеживания окончания процесса одиночного преобразования ЦАП. Т.е. запустить преобразование мы сможем, а точно определить момент его завершения – нет. Это объясняется тем, что ЦАП нужен, как правило, не для одиночных преобразований, а для генерирования аналогового сигнала с определенной частотой выборки. Для этого применяют не одиночное, а периодическое преобразование с

использованием DMA. О том, как это делается, рассказано в следующем разделе.

Возьмите мультиметр и измерьте напряжение между корпусом и сигнальной линией разъема «DAC_OUT», как показано на рисунке 4.1. Вольтметр покажет значение порядка 1 В. Это и есть значение напряжения, сформированного ЦАП. Подробности об измерении напряжения мультиметром изложены в разделе 3.3.

Внимание! Производя измерения, следите за тем, чтобы щупом мультиметра не закоротить выход ЦАП на землю. Это может привести к выходу из строя микроконтроллера.

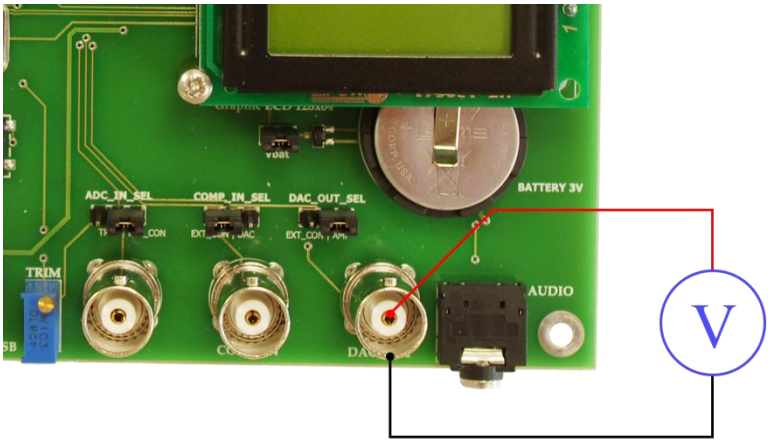


Рисунок 4.1 – Схема измерения напряжения на выходе ЦАП

Также следует отметить, что на отладочной плате имеется небольшой усилитель низкой частоты (УНЧ), на вход которого можно подать напряжение, формируемое ЦАП. К выходу усилителя (разъем «AUDIO») можно подключить обычные наушники. Это может оказаться полезным, если требуется сгенерировать сигнал звуковой частоты, например, при воспроизведении оцифрованных звукозаписей. Чтобы выход ЦАП оказался соединенным с входом УНЧ, следует установить перемычку «DAC_OUT_SEL» в положение «AMP».

4.6. Генерации аналогового сигнала заданной формы с помощью ЦАП и прямого доступа к памяти

Совместно с ЦАП можно использовать прямой доступ к памяти (DMA). Как правило, это делается в случае, если требуется генерировать аналоговый сигнал определенной формы. Поскольку такая задача применительно к ЦАП является наиболее типичной, можно считать, что режим работы ЦАП совместно с DMA является основным.

Для знакомства с таким подходом откроем в среде Keil пример `Lab4_2`, построим этот проект и загрузим программу в микроконтроллер. Как уже отмечалось в начале, в этом примере с помощью ЦАП выполняется генерирование аналогового сигнала синусоидальной или пилообразной формы. По умолчанию – синусоидальной. Сигнал получают на выводе PE0.

Процесс генерации сигнала выглядит следующим образом:

1. Организуется массив в оперативной памяти или во флеш-памяти программ, в который заносят значения требуемой периодической функции, взятые с определенным шагом. Значения функции должны укладываться в диапазон возможных входных значений ЦАП. Совокупность значений должна охватывать весь период функции.

2. Инициализируется DMA для совместной работы с ЦАП.

3. Инициализируется ЦАП для совместной работы с DMA.

4. Инициализируется аппаратный таймер, который обеспечит периодическое преобразование через заданные интервалы времени.

5. Выполнение генерации сигнала. После каждого срабатывания таймера, DMA будет считывать значение очередного элемента из массива с отсчетами функции сигнала и передавать это значение на вход ЦАП. ЦАП будет автоматически производить преобразование. Как только все элементы массива будут переданы (период функции будет завершен), DMA автоматически вернется к начальному элементу массива и весь цикл повторится заданное число раз. В этом процессе ядро микроконтроллера уже не участвует. Все работает само собой. За один цикл генерируется один период сигнала.

6. Обработка аппаратного прерывания от DMA. Как только заданное количество циклов передачи выполнено, возникает аппаратное прерывание

от DMA. В обработчике прерывания можно заново запустить работу DMA, сделав процесс генерации сигнала бесконечным.

Параметры функции сигнала задаются в заголовке `generator.h`. Для всех видов сигналов указываются:

- размер буфера отсчетов `U_GENERATOR_BUFFER_SIZE`;
- частота `U_GENERATOR_F`;
- постоянная составляющая `U_GENERATOR_C`.

Для синусоидального сигнала дополнительно указывается амплитуда синусоиды `U_GENERATOR_SIN_A`. Для пилообразного сигнала – амплитуда пилы `U_GENERATOR_TRIANGLE_A`.

Размер буфера отсчетов определяет количество отсчетов за период функции, то есть, сколько раз ЦАП будет выполнять преобразование за один период функции. С помощью ЦАП любой сигнал аппроксимируется ступенчатым сигналом. Чем больше буфер, тем точнее аппроксимируется функция, так как размер ступенек будет меньше. Однако при этом менее точно удается выдержать требуемую частоту сигнала. Поэтому размер буфера выбирают разумный: 16…256 отсчетов.

Вычисление значений отсчетов функции сигнала и занесение их в соответствующий буфер производится при вызове функции `U_Generator_Init` вскоре после запуска микроконтроллера. При этом для синусоидального сигнала вызывается функция `Function_Sin_Setup`.

Для того чтобы увидеть генерируемый сигнал, необходимо использовать осциллограф. Далее будет рассказано, как это делается.

4.7. Основы работа с осциллографом

С помощью осциллографа можно визуально наблюдать зависимость напряжения от времени в требуемой электрической цепи, другими словами, видеть форму электрического сигнала. Такая возможность очень упрощает отладку электронных устройств, в том числе и отладку программного обеспечения микроконтроллеров. По сути, осциллограф является вольтметром, который представляет результаты измерений в виде графика зависимости напряжения от времени.

В качестве осциллографа будем использовать осциллограф-приставку к ПК, выпускаемый в Одессе (рисунок 4.2). Этот осциллограф представляет собой

небольшую пластиковую трубку-корпус, к которой подключается USB-кабель для связи с ПК, и щуп в виде иглы и общего провода с зажимом «крокодил». Осциллограф измеряет напряжение между общим проводом и иглой. Общий провод обычно присоединяют к заземленной поверхности, а иглой прикасаются к исследуемой цепи.

Для отображения информации осциллограф подключается к ПК, на котором должны быть установлены драйвер осциллографа и специальная программа Oscill. Программа выполняет отображение исследуемого сигнала на мониторе ПК.

Рисунок 4.2 – Универсальный осциллограф-приставка к ПК

Внимание! Игла у щупа осциллографа весьма острая, поэтому соблюдайте осторожность в работе. Особенно берегите глаза!

Прикрутите к осциллографу щуп с иглой, присоедините к осциллографу кабель и подключите кабель к USB-разъему компьютера. Запустите программу Oscill. На экране появится главное окно программы (рисунок 4.3).

Основную часть окна занимает черный экран осциллографа. На этом экране в виде зеленой линии, называемой **осциллограмма**, показывается зависимость напряжения в исследуемой электрической цепи от времени. По вертикали располагается ось напряжения, а по горизонтали – ось времени. Для удобства восприятия на экран нанесена сетка с одинаковыми

квадратными ячейками. С помощью этой сетки нетрудно измерять длительность нужного участка сигнала, а также значение напряжения.

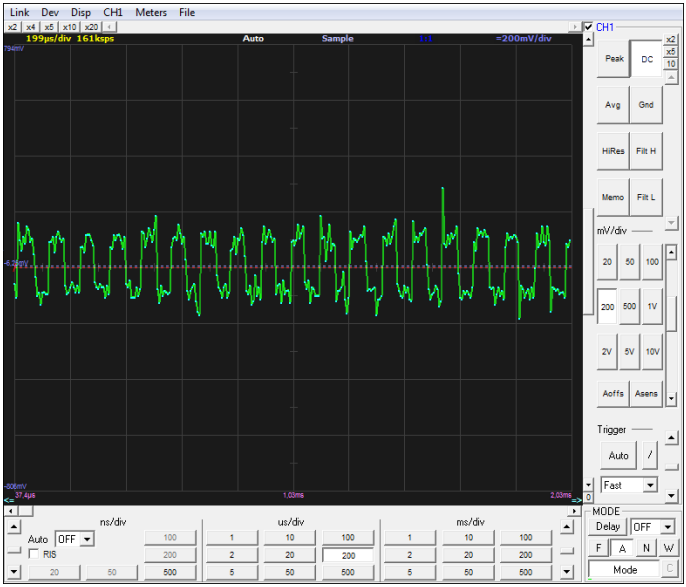


Рисунок 4.3 – Главное окно программы Oscill

Основными настройками любого осциллографа являются:

- скорость развертки (мс/клетку, мкс/клетку)
- усиление сигнала (В/клетку, мВ/клетку);
- вид пропускаемого сигнала: переменный или переменный и постоянный;
- уровень запуска (В или мВ);
- режим синхронизации;
- положение по вертикали.

Скорость развертки (рисунок 4.4) задает, сколько микро- или миллисекунд приходится на одну клетку экрана по горизонтали. Как видно из рисунка 4.4, для данного осциллографа можно задать скорость развертки от 1 мкс/клетку (us/div) до 500 мс/клетку (ms/div). Переключение скорости развертки производится нажатием на соответствующие кнопки панели инструментов.

Скорость развертки нужно выбирать исходя из того, какой частоты будет исследуемый сигнал. Так, если исследуется сигнал частотой 250 Гц (период сигнала равен 1 / 250 Гц = 4 мс), целесообразно выбрать скорость развертки порядка 2 мс/клетку. Это позволит изобразить период сигнала в двух клетках по горизонтали. На экране уместится 5 периодов сигнала. Для сигнала частотой 10 КГц (период сигнала равен 1 / 10000 Гц = 100 мкс), целесообразно выбрать скорость развертки порядка 50 мкс/клетку.

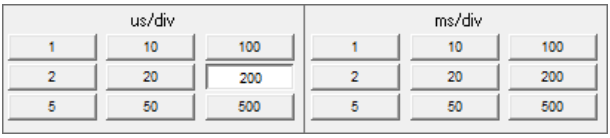


Рисунок 4.4 – Настройка скорости развертки осциллографа

Усиление сигнала (рисунок 4.5) задает, сколько вольт или милливольт приходится на одну клетку по вертикали. Для данного осциллографа можно задать усиление сигнала от 20 мВ/клетку до 10 В/клетку.

Рисунок 4.5 – Настройка усиления сигнала

Усиление выбирают исходя из размаха исследуемого сигнала, стараясь, чтобы сигнал по вертикали занимал почти весь экран, но не выходил за его пределы. Так, если размах сигнала составляет 3 В, целесообразно выбрать усиление 500 мВ/клетку. В таком случае сигнал

займет по вертикали 6 клеток. Если выбрать усиление 200 мВ/клетку, то сигнал уже не уместится на экране.

Вид пропускаемого сигнала (рисунок 4.6) задает, какой сигнал мы сможем исследовать: только переменный (AC) или переменный и постоянный (DC). Чаще используют режим DC, в котором можно увидеть постоянную составляющую сигнала.

В режиме AC постоянная составляющая сигнала не будет пропускаться. Это может оказаться полезным, если постоянная составляющая сигнала велика по сравнению с переменной составляющей, а нужно тщательно разглядеть переменную составляющую.

Мы будем использовать режим DC. Для этого нажмем одноименную кнопку на панели, как показано на рисунке 4.6.

Рисунок 4.6 – Настройка пропускаемого сигнала

Уровень запуска (рисунок 4.7) задает значение напряжения, с которого будет запускаться развертка. Изображается на экране в виде прерывистой красной горизонтальной линии. Правильный выбор уровня запуска позволяет сделать изображение осциллограммы стабильным, не срывающимся. Если правильный уровень запуска не подобрать, развертка будет «бегать» по горизонтали, и исследовать сигнал будет невозможно. Уровень запуска подбирают с помощью вертикальной полосы прокрутки,

показанной на рисунке 4.7. Если двигать скроллер, будет двигаться и красная линия уровня запуска развертки.

Рисунок 4.7 – Настройка уровня запуска

Режим синхронизации (рисунок 4.8) задает один из возможных способов синхронизации развертки. Для исследования периодических сигналов следует нажать кнопку Mode, а затем одну из кнопок – F, A, N либо W. Разница между ними следующая:

- F – свободная развертка (Free): не ждет синхронизации, стартует сразу после окончания предыдущей развертки;
- A – автоматический запуск развертки (Auto): ждет синхронизации, но если ее нет, все равно начинает новую развертку;
- N – ждущий запуск развертки: ждет синхронизации заданное время, а если ее нет, то новая развертка не начинается;
- W – бесконечное ожидание развертки.

По умолчанию будем использовать режим A.

Рисунок 4.8 – Выбор режима синхронизации

Положение по вертикали позволяет задать положение на экране уровня нуля вольт. Этот уровень изображен прерывистой горизонтальной прямой синего цвета. Перемещая уровень нуля вольт с помощью вертикального скроллера, расположенного справа от экрана, мы, по сути, перемещаем вверх или вниз изображение сигнала.

Весьма полезной возможностью является отображение оценки частоты исследуемого сигнала. Для этого следует выбрать пункт меню *Meters – Frequency*. В результате появится отдельное окошко с числовым значением частоты (рисунок 4.9).

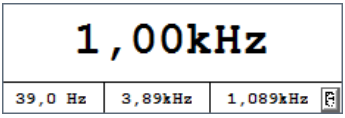


Рисунок 4.9 – Окно со значением частоты сигнала

Испытаем работу осциллографа, прикоснувшись пальцем к игле щупа (достаточно сбоку). На экране осциллографа появится изображение электромагнитных наводок на ваше тело.

Таким нехитрым способом обычно проверяют работоспособность осциллографа. Если осциллограф реагирует на прикосновение к телу, значит, он исправен. Если наводки не отображаются, значит, осциллограф не настроен должным образом или вышел из строя.

Конечно, у осциллографа, особенно цифрового, есть еще множество иных настроек, но с ними мы будем постепенно знакомиться в дальнейшем.

Теперь приступим к исследованию сигнала, генерируемого с помощью ЦАП и DMA. Если это еще не сделано, откройте в среде Keil проект `Lab4_2`, постройте его и загрузите программу в микроконтроллер. Отладочная плата, естественно, должна быть подключена к ПК.

Задайте скорость развертки 500 мкс/клетку, усиление – 200 мВ/клетку. Уровень нуля вольт переместите на одну клетку выше нижнего края экрана.

Подключите зажим «крокодил» общего провод осциллографа к цепи GND (земля) отладочной платы. Лучше всего для этого использовать выступ на корпусе разъема «DAC_OUT». Щуп-иглу прижмите к сигнальному контакту этого же разъема.

Внимание! Действуйте при этом осторожно, чтобы не замкнуть иглой сигнальную линию на корпус разъема. Это может вывести микроконтроллер из строя!

На экране появится изображение синусоиды. Если синусоида «бегает» по экрану, выставите уровень запуска так, чтобы он находился в пределах изображения синусоиды, лучше ближе к верхнему краю. Синусоида «замрет» на месте. При этом говорят, что достигнута синхронизация сигнала.

В окошке с измерением частоты увидим показание порядка 1 КГц, что и должно быть, так как генерируется сигнал частотой 1000 Гц.

Задайте в заголовке `generator.h` размер буфера отсчетов, равный 16, скорость развертки осциллографа установите 200 мкс/клетку. Получите осциллограмму сигнала на разъеме «DAC_OUT» отладочной платы. Вы увидите грубую синусоиду, составленную из больших ступенек (рисунок 4.10).

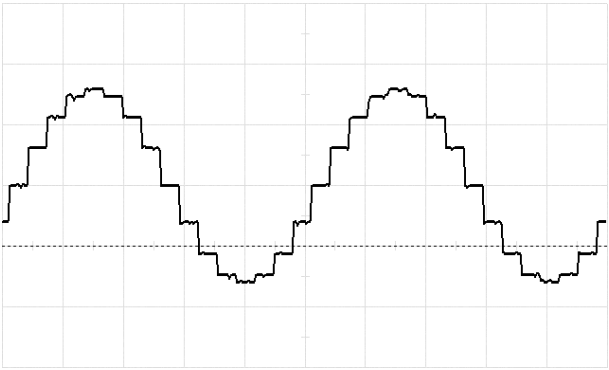


Рисунок 4.10 – Гармонический сигнал при малом количестве отсчетов за период

Теперь поменяйте размер буфера отсчетов на 256 и вновь получите осциллограмму сигнала. Синусоида станет практически гладкой (рисунок 4.11).

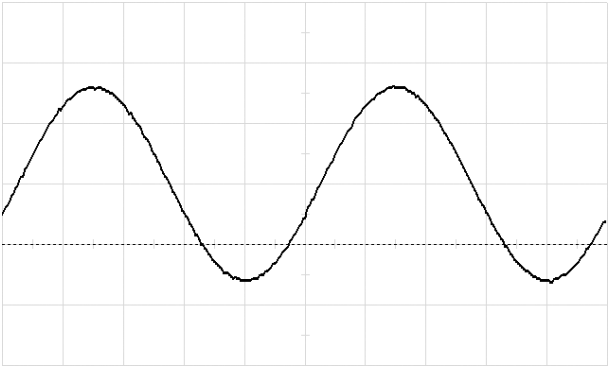


Рисунок 4.11 – Гармонический сигнал при большом количестве отсчетов за период

Далее измерьте период сигнала и его удвоенную амплитуду (размах). Делается это так: нажав левую кнопку мыши, выделяем нужный интервал по горизонтали (при измерении времени) или по вертикали (при измерении

напряжения). Результаты измерения отображаются на экране белыми линиями и символами. На рисунке 4.12 показаны результаты измерения периода (1 мс), а на рисунке 4.13 – результаты измерения размаха сигнал (634 мВ).

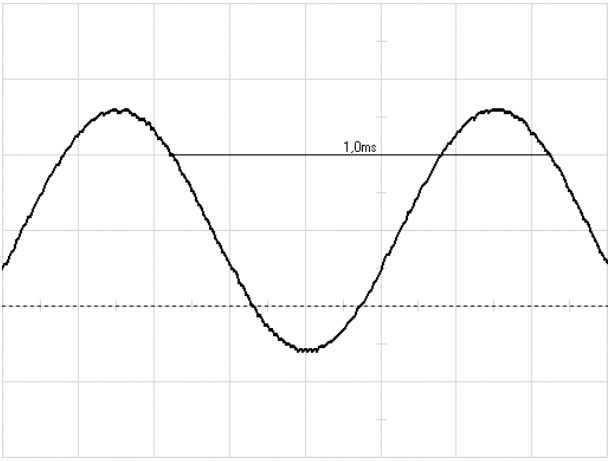


Рисунок 4.12 – Результат измерения периода сигнала

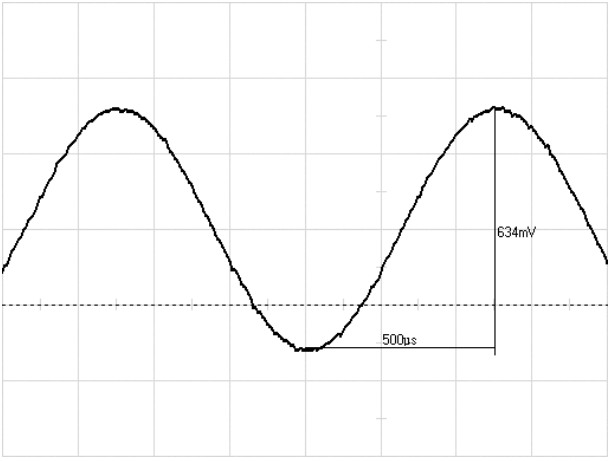


Рисунок 4.13 – Результат измерения размаха сигнала

4.8. Настройка прямого доступа к памяти для работы с ЦАП

Рассмотрим особенности настройки DMA при совместном использовании с ЦАП. Совокупность настроек DMA в целом располагаем в структуре DMA_InitStructure:

```
DMA_CtrlDataInitTypeDef DMA_InitStructure;
```

Совокупность настроек канала DMA располагаем в структуре DMA_Channel_InitStructure:

```
DMA_ChannelInitTypeDef DMA_Channel_InitStructure;
```

Заметим, что обе структуры являются глобальными, поскольку с ними придется работать и за пределами модуля generator.c, а именно в модуле обработчиков аппаратных прерываний MDR32F9Qx_it.c.

Настройки DMA реализованы в функции DMA_Config, расположенной в модуле generator.c. Поскольку в предыдущей работе было подробно рассказано про основные настройки DMA, здесь остановимся лишь на некоторых особенностях.

Разрешаем тактирование DMA:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_DMA | RST_CLK_PCLK_SSP1 |
                 RST_CLK_PCLK_SSP2, ENABLE);
```

При этом нам необходимо также разрешить тактирование интерфейсов SSP1 и SSP2. Это позволяет исправить застарелую ошибку в работе DMA для микроконтроллеров семейства 1986BE9x. Если этого не сделать, нормальной работы DMA не получится.

По той же причине временно запрещаем все аппаратные прерывания, включая SSP1 и SSP2:

```
NVIC->ICPR[0] = 0xFFFFFFFF;
NVIC->ICER[0] = 0xFFFFFFFF;
```

Деинициализируем (сбрасываем настройки) DMA:

```
DMA_DeInit();
```

Инициализируем структуру для настройки DMA, передав в качестве параметра адрес структуры для настройки требуемого канала DMA:

```
DMA_StructInit (&DMA_Channel_InitStructure);
```

Далее заполняем структуру для настройки DMA.

Задаем базовый адрес регистра данных ЦАП. Туда будут записываться данные с помощью DMA.

```
DMA_InitStructure.DMA_DestBaseAddr = (uint32_t)(&(MDR_DAC->DAC2_DATA));
```

Указываем адрес начала буфера отсчетов сигнала `Signal`. Из него будут считываться данные:

```
DMA_InitStructure.DMA_SourceBaseAddr = (uint32_t) Signal;
```

Задаем размер буфера отсчетов сигнала. Столько раз DMA будет производить считывание данных:

```
DMA_InitStructure.DMA_CycleSize = U_GENERATOR_BUFFER_SIZE;
```

Запрещаем автоматическое увеличение адреса для приемника, поскольку пишем данные всегда в один и тот же регистр данных ЦАП:

```
DMA_InitStructure.DMA_DestIncSize = DMA_DestIncNo;
```

Разрешаем автоматическое увеличение адреса в памяти на полслова (т.е. на 2), поскольку считываем данные из разных ячеек буфера отсчетов:

```
DMA_InitStructure.DMA_SourceIncSize = DMA_SourceIncHalfword;
```

Указываем, по сколько байт будем передавать за один раз (по два байта или по полслова):

```
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
```

Указываем, сколько раз будем повторять полные циклы передачи данных:

```
DMA_InitStructure.DMA_NumContinuous = DMA_Transfers_32;
```

Возьмем, к примеру, 32 раза. Это означает, что DMA 32 раза подряд выполнит передачу по U_GENERATOR_BUFFER_SIZE элементов данных, а затем остановится. Можно выбирать значения: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 и 1024. Режим бесконечного повтора передачи, к сожалению, не предусмотрен.

Включаем режимы защиты источника и приемника:

```
DMA_InitStructure.DMA_SourceProtCtrl = DMA_SourcePrivileged;
DMA_InitStructure.DMA_DestProtCtrl = DMA_DestPrivileged;
```

Задаем базовый режим работы DMA:

```
DMA_InitStructure.DMA_Mode = DMA_Mode_Basic;
```

Отметим также, что при работе DMA совместно с ЦАП нередко используется режим пинг-понг (DMA_Mode_PingPong), при котором можно поочередно брать значения для ЦАП из двух источников. Это позволяет за время работы с одним источником подготовить данные в другом источнике. Такой подход очень полезен при генерации апериодических сигналов, например, при воспроизведении звукозаписей. Пока небольшой фрагмент воспроизводится из одного массива, другой фрагмент подготавливается в другом массиве. Затем – наоборот. Но об этом поговорим в иной раз.

Теперь же заполним структуру для настройки канала DMA.

К первичному блоку настроек канала привяжем ранее заполненную структуру настройки DMA и выберем ее в качестве используемой в данный момент:

```
DMA_Channel_InitStructure.DMA_PriCtrlData = &DMA_InitStructure;
DMA_Channel_InitStructure.DMA_SelectDataStructure =
                                                DMA_CTRL_DATA_PRIMARY;
```

Вообще-то есть два блока настроек канала DMA: первичный (DMA_PriCtrlData) и альтернативный (DMA_AltCtrlData). Альтернативный требуется лишь при режиме пинг-понг, поэтому нами сейчас не рассматривается.

Приоритет канала DMA установим по умолчанию:

```
DMA_Channel_InitStructure.DMA_Priority = DMA_Priority_Default;
```

Снимем запрет на одиночные запросы к DMA:

```
DMA_Channel_InitStructure.DMA_UseBurst = DMA_BurstClear;
```

Полученной структурой инициализируем канал DMA для работы с таймером TIMER1:

```
DMA_Init (DMA_Channel_TIM1, &DMA_Channel_InitStructure);
```

Снимем запрет на запросы к DMA со стороны таймера TIMER1. Это рекомендуется для нормальной работы DMA (что-то типа заплатки для устранения аппаратных ошибок в микроконтроллерах 1986BE9x):

```
MDR_DMA->CHNL_REQ_MASK_CLR = 1 << DMA_Channel_TIM1;
MDR_DMA->CHNL_USEBURST_CLR = 1 << DMA_Channel_TIM1;
```

Разрешим работу DMA с каналом для таймера TIMER1

```
DMA_Cmd (DMA_Channel_TIM1, ENABLE);
```

Зададим приоритет 1 для аппаратного прерывания от DMA:

```
NVIC_SetPriority (DMA_IRQn, 1);
```

Также необходимо настроить аппаратный таймер, задающий период, через который DMA передает данные на ЦАП. Эти настройки реализованы в функции `TIM_Config` модуля `generator.c`.

Настройки таймера размещают в структуре `TIM_CntInit`:

```
TIMER_CntInitTypeDef TIM_CntInit;
```

В нашем случае будем использовать таймер TIM1. Сбросим его настройки и включим тактирование:

```
TIMER_DeInit (MDR_TIMER1);
RST_CLK_PCLKcmd (RST_CLK_PCLK_TIMER1, ENABLE);
```

Настроим частоту тактирования таймера, задав делитель частоты импульсов, поступающих на него, равным 1:

```
TIMER_BRGInit (MDR_TIMER1, TIMER_HCLKdiv1);
```

Таким образом, на таймер будет поступать частота, равная частоте процессорного ядра, т.е. 80 МГц. При желании можно выбрать коэффициент деления, равный: 1, 2, 4, 8, 16, 32, 64 или 128.

Теперь заполняем данными структуру для настройки таймера.

Период таймера устанавливаем в зависимости от требуемой частоты сигнала:

```
TIM_CntInit.TIMER_Period = U_GENERATOR_PERIOD (U_GENERATOR_F);
```

Период рассчитывается с помощью макроса `U_GENERATOR_PERIOD(F)`, определенного в заголовке `generator.h`. Макросы языка в Си позволяют выполнить вычисления на этапе компиляции проекта, не занимая затем процессорного времени. Вот как выглядит определение этого макроса:

```
#define U_GENERATOR_PERIOD(F) ((uint16_t)((uint32_t)(80000000) /
                    (uint32_t)(F * U_GENERATOR_BUFFER_SIZE)) - 1)
```

Здесь реализована формула (4.4), вычисляющая период P в тактах таймера по заданной частоте сигнала F в Гц и размере буфера отсчетов S:

$$P = \frac{80000000}{F \cdot S} - 1 \qquad (4.4)$$

Число 80000000 соответствует частоте тактирования таймера в Гц. В нашем случае она и равна 80 МГц.

Задаем предделитель импульсов. Он позволяет еще раз поделить частоту импульсов, тактирующих таймер. Но нам и этого не надо: значение 0 соответствует делению на единицу:

```
TIM_CntInit.TIMER_Prescaler = 0;
```

В результате таймер будет считать импульсы частотой 80 МГц.

Направление счета задаем вверх (таймер/счетчик будет увеличиваться):

```
TIM_CntInit.TIMER_CounterDirection = TIMER_CntDir_Up;
```

Направление счета делаем фиксированным, т.е. все время будем считать вверх:

```
TIM_CntInit.TIMER_CounterMode = TIMER_CntMode_ClkFixedDir;
```

Назначение остальных полей структуры сейчас не так важно, поэтому оставим их без рассмотрения.

Инициализируем таймер заполненной структурой:

```
TIMER_CntInit (MDR_TIMER1, &TIM_CntInit);
```

Разрешаем работу DMA совместно с таймером TIMER1:

```
TIMER_DMACmd (MDR_TIMER1, TIMER_STATUS_CNT_ARR, ENABLE);
```

При этом запрос к DMA будет возникать всякий раз при достижении таймером/счетчиком значения, заданного в регистре сравнения ARR, т.е. когда таймер отсчитает заданный временной промежуток. По сути, этой командой мы привязываем DMA к таймеру 1.

Далее разрешаем работу таймера TIMER1:

```
TIMER_Cmd (MDR_TIMER1, ENABLE);
```

И, наконец, разрешаем аппаратные прерывания от DMA.

С этого момента сигнал начинает генерироваться. Причем процессорное ядро на этот процесс будет отвлекаться лишь изредка – при возникновении прерывания по окончании 32 циклов передачи данных.

В эти моменты программа микроконтроллера будет автоматически переключаться на подпрограмму-обработчика аппаратного прерывания от DMA, определенного в модуле `MDR32F9Qx_it.c`:

```
void DMA_IRQHandler (void)
{
   // Подготовить к работе новый цикл цифро-аналоговых преобразований
   DMA_InitStructure.DMA_CycleSize = U_GENERATOR_BUFFER_SIZE;
   DMA_Init (DMA_Channel_TIM1, &DMA_Channel_InitStructure);
}
```

В обработчике вновь уточняется объем передаваемых за один цикл данных и заново инициализируется канал DMA. После этого генерация сигнала продолжается.

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 4.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. В проекте `Lab4_1` подберите значение, подаваемое на вход ЦАП, таким образом, чтобы на выходе ЦАП получилось напряжение 2,75 В. Результаты продемонстрируйте с помощью мультиметра или осциллографа.

Пояснение. Константа `U_DAC_VALUE`, задающая входное значение, содержится в заголовке `dac.h`.

2. Используя проект `Lab4_2`, сформируйте на выходе ЦАП периодический сигнал синусоидальной формы с частотой 2 КГц, амплитудой 500 мВ и постоянной составляющей 1200 мВ. При этом используйте разное количество отсчетов на период: 32, 128, 256. Результаты продемонстрируйте с помощью осциллографа.

Пояснение. Все требуемые параметры находятся в заголовке `generator.h`.

3. Используя проект `Lab4_2`, сформируйте на выходе ЦАП периодический сигнал пилообразной формы с частотой 800 Гц, амплитудой 1500 мВ и постоянной составляющей 50 мВ. Количество отсчетов возьмите 256. Результаты продемонстрируйте с помощью осциллографа.

Пояснение. Форма генерируемого сигнала задается в функции `U_Generator_Init` модуля `generator.c`.

Контрольные вопросы

1. Что такое ЦАП? Для чего он нужен?
2. Какие основные настройки следует делать при использовании осциллографа?
3. Как влияет разрядность ЦАП на количество возможных уровней формируемого выходного напряжения?
4. Как запустить одиночное цифро-аналоговое преобразование?
5. Для чего используют DMA совместно с ЦАП?
6. Какие основные настройки требуется произвести при инициализации ЦАП?
7. Какие основные настройки следует выполнить при использовании цифрового осциллографа-приставки?
8. Как с помощью подручных средств проверить, работает ли осциллограф?

Глава 5

Широтно-импульсная модуляция

Цель работы: получение навыков использования широтно-импульсной модуляции для регулирования мощности, подводимой к нагрузке.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- периферийный модуль;
- цифровой осциллограф-приставка Oscill;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- драйвер для осциллографа-приставки Oscill;
- приложение Oscill для осциллографа-приставки Oscill;
- примеры кода программ.

5.1. Подготовка к работе

Отключите питание отладочной платы, если оно было подключено, и с помощью перемычек подключите к ней периферийный модуль согласно таблице 5.1.

Таблица 5.1 – Подключение периферийного модуля к отладочной плате

№ п/п	Цвет провода	Периферийный модуль		Отладочная плата	
		Имя штыря	Имя разъема	Имя штыря	Имя разъема
1	черный	GND1	XP1	1	X27
2	синий	средний	XP1	25	X27
3	красный	+5V	XP1	27	X27

Запустите среду Keil μVision и откройте проект, расположенный по адресу `Samples\Project\Lab5_1\MDK-ARM`. Подключите к отладочной плате программатор и питание (подробности в разделе 1.5). Проверьте правильность настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

5.2. Описание проектов

В примере `Lab5_1` показано, как задавать определенную мощность, подводимую к лампе, за счет формирования на выводе PF6 цифрового сигнала с широтно-импульсной модуляцией с параметрами, фиксировано задаваемыми в программе. На ЖКИ выводится коэффициент заполнения импульсов, питающих лампу, выраженный в процентах.

В примере `Lab5_2` реализует то же самое, но с возможностью плавно изменять яркость горения лампы поворотом ручки потенциометра.

5.3. Понятие широтно-импульсной модуляции

Широтно-импульсной модуляцией (ШИМ, или, по-английски, PWM – Power Width Modulation) называют способ регулирования среднего напряжения, подаваемого на нагрузку, за счет изменения скважности импульсов [13].

Скважностью называют отношение периода импульса к его длительности. Это безразмерная величина, определяемая по формуле (5.1):

$$S = \frac{T}{\tau}, \tag{5.1}$$

где T – период импульсов, τ – длительность импульсов, выраженные в одинаковых единицах измерения времени (например, в микросекундах или миллисекундах).

Чем короче длина импульса относительно его периода, тем выше скважность. Так, при S = 2, длительность импульса составляет половину периода, а при S = 10 – лишь 10% от периода. Очевидно, что скважность не может быть меньше единицы. На рисунке 5.1 показаны импульсы с высокой скважностью, а на рисунке 5.2 – с низкой.

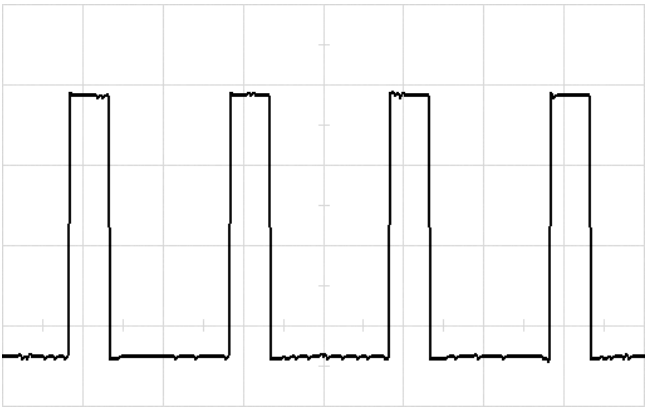


Рисунок 5.1 – Прямоугольные импульсы с высокой скважностью

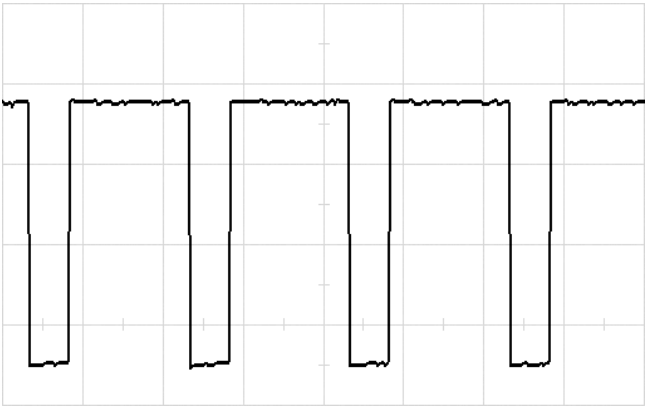


Рисунок 5.2 – Прямоугольные импульсы с низкой скважностью

Величина, обратная скважности, называется **коэффициентом заполнения** и может быть определена по формуле (5.2):

$$D = \frac{1}{S} \tag{5.2}$$

Коэффициент заполнения показывает, в течение какой части периода импульса сигнал является активным. Коэффициент заполнения нередко выражают в процентах (5.3):

$$D = \frac{\tau}{T} \cdot 100\% \tag{5.3}$$

С помощью ШИМ можно легко и рационально регулировать среднюю электрическую мощность, подводимую к некоторой нагрузке. Нагрузка подключается к источнику напряжения на небольшое время, а затем отключается от него также на некоторое время. Этот процесс периодически повторяется. В результате действующее значение напряжения, питающего нагрузку, определяется выражением (4):

$$\overline{U} = \frac{U_{\max}}{\sqrt{S}}, \tag{5.4}$$

где $U_{\max}$ – амплитуда импульсов, питающих нагрузку, S – скважность импульсов. Позвольте не загромождать книгу формулами с интегралами, с помощью которых можно это доказать. Действующее значение напряжения вы сможете увидеть на вольтметре, подключенном к нагрузке.

Если нагрузка обладает активным электрическим сопротивлением R, то действующая мощность, потребляемая нагрузкой, составит (вспомним закон Ома):

$$\bar{P} = \frac{\overline{U}^2}{R} = \frac{U_{\max}^2}{R \cdot S} \tag{5.5}$$

Таким образом, мощность, подводимая к нагрузке, обратно пропорциональна скважности питающих импульсов. Меняя скважность, можно менять мощность на нагрузке в пределах от 0 до 100%. Изменять скважность очень просто, если использовать в качестве формирователя ШИМ микроконтроллер. Это доступно программным образом.

ШИМ применяют для регулирования яркости горения осветительных ламп, скорости вращения двигателей, температуры нагрева теплоэлектронагревателей и т.д.

Достоинствами ШИМ являются очень высокий КПД и относительная простота реализации (на современной элементной базе).

В нашей лабораторной работе используется простейшая схема, реализующая ШИМ для управления яркости горения лампы накаливания (рисунок 5.3).

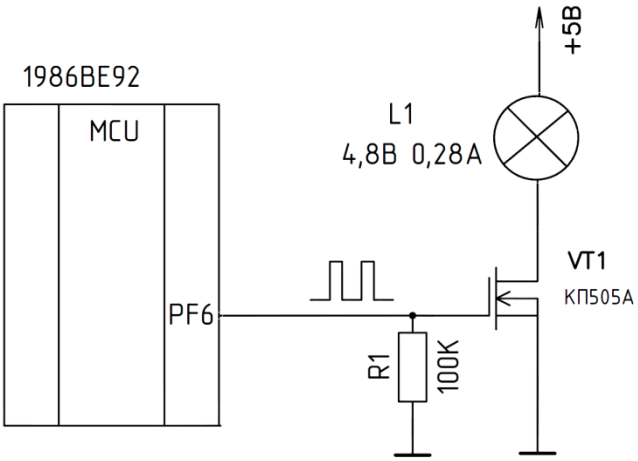


Рисунок 5.3 – Принципиальная схема
подключения лампы накаливания к микроконтроллеру

Для управления лампой накаливания используется N-канальный полевой транзистор. Его приходится применять по той причине, что к выходам микроконтроллера нельзя непосредственно подключать мощную нагрузку. Лампа накаливания потребляет ток порядка 300 мА, что во много раз превышает допустимый ток на цифровом выходе микроконтроллера.

Вспомним (или впервые рассмотрим ☺), как называются выводы полевого транзистора и как он работает.

На рисунке 5.4 показано, как обозначается на схемах N-канальный полевой транзистор с изолированным затвором и индуцированным каналом. В зарубежной литературе такой транзистор называют MOSFET (Metal–Oxide–Semiconductor Field-Effect Transistor). Не станем сейчас рассматривать классификацию полевых транзисторов (существует много разных типов), просто запомним, что для реализации ШИМ чаще всего используют транзисторы именно такого типа.

У полевого транзистора есть два силовых вывода – сток и исток. Их можно рассматривать, как два вывода у обычного выключателя. Также есть управляющий вывод – затвор. Затвор управляет работой транзистора: напряжением, приложенным к затвору относительно истока, можно изменять ток, протекающий между истоком и стоком.

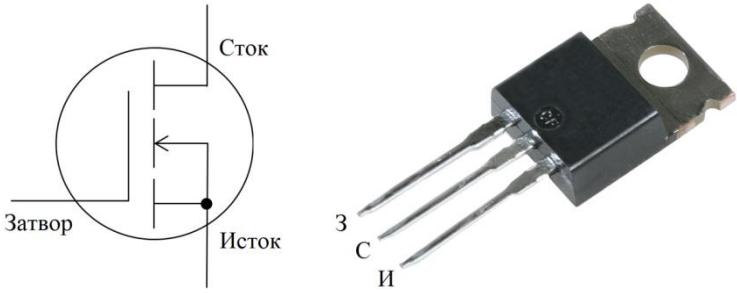


Рисунок 5.4 – N-канальный полевой транзистор
с изолированным затвором и индуцированным каналом

Очень часто полевой транзистор используют в ключевом режиме. Ключевой режим означает, что транзистор либо полностью закрыт (внутренне сопротивление канала транзистора очень велико – десятки мегаом, и ток через него практически не течет), либо полностью открыт (внутренее сопротивление канала транзистора очень мало – доли ома, и ток через него течет беспрепятственно).

В нашем случае используется отечественный транзистор КП505А. Такой транзистор полностью закрыт, если подавать на его затвор управляющее напряжение менее +0,5 В относительно истока, что соответствует напряжению логического нуля микроконтроллера. Транзистор полностью открыт, если подавать на его затвор управляющее напряжение более +2,0 В относительно истока, что соответствует напряжению логической единицы микроконтроллера.

Согласно рисунку 5.3, транзистор VT1 работает в ключевом режиме, управляя лампой накаливания L1. На затвор подаются импульсы ШИМ, формируемые микроконтроллером. В момент, когда на затвор поступает напряжение логической единицы (примерно +3 В), транзистор быстро (но не

мгновенно!) открывается. А когда напряжение логического нуля (примерно +0,1 В) – быстро (но тоже не мгновенно!) закрывается. Таким образом, к лампе то подключается, то отключается источник напряжения.

На рисунках 5.1 и 5.2 были как раз показаны осциллограммы импульсов, питающих лампу.

Резистор R1, притягивающий затвор к истоку, нужен для того, чтобы не было ложных срабатываний ключа. Полевой транзистор управляется напряжением, приложенным к затвору, ток же через затвор практически отсутствует. Поэтому полевой транзистор может случайно открыться от электромагнитной помехи или статического заряда. Резистор же подавляет (шунтирует) эти помехи.

Почему при использовании ШИМ достигается высокий КПД? Это определяется тем, что транзисторный ключ, используемый для коммутации тока в нагрузке, рассеивает на себе очень малую мощность. Когда ключ закрыт, ток через него практически не течет, следовательно, и мощность не потребляется. Когда же ключ открыт, его сопротивление очень мало, следовательно, и напряжение, падающее на ключе, также очень мало (доли вольта). Поэтому коэффициент полезного действия будет высок.

Так, используемый нами транзистор КП505А в открытом состоянии имеет внутреннее сопротивление порядка 0,5 Ом. При токе через лампочку, равном 280 мА имеем (по закону Ома) падение напряжения на ключе:

$$U_{\text{ключа}} = I \cdot R_{\text{ключа}} = 0{,}5\ \text{Ом} \cdot 0{,}28\ \text{А} = 0{,}14\ \text{В} \qquad (5.6)$$

Мощность, рассеиваемая на ключе, составит:

$$P_{\text{ключа}} = I \cdot U_{\text{ключа}} = 0{,}28\ \text{А} \cdot 0{,}14\ \text{В} \approx 0{,}039\ \text{Вт} \qquad (5.7)$$

Мощность же, подводимая к лампе, при напряжении питания схемы 5 В составит:

$$P_{\text{лампы}} = I \cdot (U_{\text{пит}} - U_{\text{ключа}}) = 0{,}28\ \text{А} \cdot (5\ \text{В} - 0{,}14\ \text{В}) \approx 1{,}36\ \text{Вт} \qquad (5.8)$$

КПД, выраженный в процентах, составит:

$$\eta = \frac{P_{\text{лампы}}}{P_{\text{лампы}} + P_{\text{ключа}}} \cdot 100\% = \frac{1{,}36\ \text{Вт}}{1{,}36\ \text{Вт} + 0{,}039\ \text{Вт}} \approx 97\% \qquad (5.9)$$

Таким образом, получаем высокий КПД при любой скважности импульсов, а, следовательно, и при любой средней мощности, которую мы хотим получить на нагрузке. Транзистор при этом практически не будет греться.

Не будем сейчас останавливаться на рассмотрении альтернативных (и менее рациональных) схемных и программных решений для регулирования мощности. Скажем только, что, если для этого использовать ЦАП и транзистор, работающий в активном (не ключевом) режиме, КПД составит от 3 до 97%, в зависимости от мощности, которую бы мы хотели получить на нагрузке. Если требуется получить малую мощность, то вся остальная мощность будет рассеиваться на транзисторе, и он будет греться очень сильно.

5.4. Проблема выбора частоты импульсов ШИМ

Конечно, не все так просто с применением ШИМ. Важно бывает правильно выбрать частоту импульсов.

Если частота импульсов слишком велика, то это приводит к следующим проблемам:

- дополнительные потери мощности на ключе, связанные с конечным временем открытия и закрытия ключа;
- уменьшение количества ступеней изменения коэффициента заполнения импульсов.

Первая проблема связана с тем, что между затвором и истоком транзистора всегда имеется некоторая паразитная емкость (по сути, как бы, маленький конденсатор, расположенный внутри транзистора). Из-за этой емкости транзистор не может мгновенно открыться или закрыться: сначала должна перезарядиться емкость. Пока емкость перезаряжается, транзистор временно находится в активном режиме, т.е. не полностью открыт или закрыт. В это время, обычно составляющее доли микросекунд, на транзисторе падает в среднем до половины напряжения питания, а значит, рассеивается и до половины мощности. Это время называют временем переходного процесса. Пока частота мала (до нескольких сотен КГц), время

переходного процесса составляет лишь незначительную часть от ширины и, тем более, периода импульса. Но при больших частотах переходной процесс займет значительную часть импульса, что приведет к большой потере мощности на ключе. КПД упадет, а транзистор будет сильно греться.

На рисунках 5.5 и 5.6 показаны осциллограммы импульсов, питающих лампу при разных частотах, но одинаковом коэффициенте заполнения 20%.

На рисунке 5.5 частота составляет 10 КГц. Видно, что импульсы близки к прямоугольной форме: время открытия и закрытия ключа очень мало по сравнению с шириной импульса. Фронты импульсов практически вертикальные.

На рисунке 5.6 частота составляет 1 МГц. Форма импульсов далека от прямоугольной: верх импульса практически выродился в острый угол. Фронты импульсов сильно наклонены. КПД в этом случае составит лишь порядка 50%. Если теперь немного уменьшить коэффициент заполнения, то ключ попросту перестанет открываться: напряжение на затворе транзистора не будет успевать подниматься до порога открытия транзистора. То же самое произойдет, если еще увеличить частоту импульсов при той же скважности.

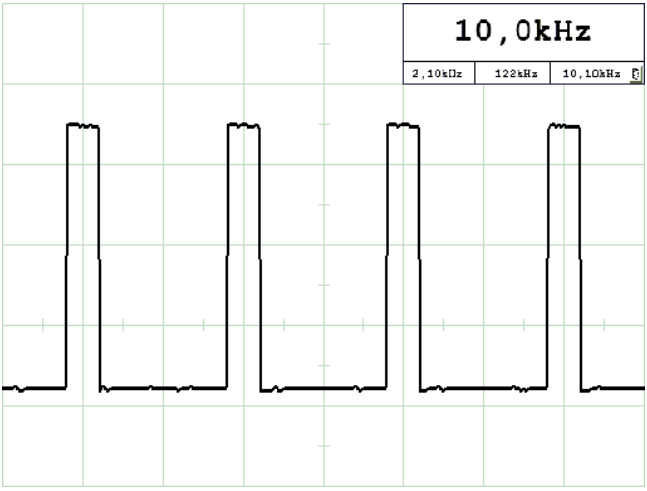


Рисунок 5.5 – Импульсы, питающие лампу при частоте 10 КГц

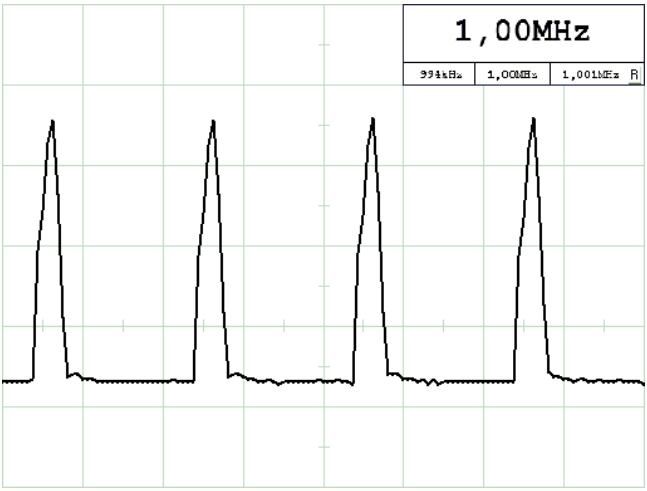


Рисунок 5.6 – Импульсы, питающие лампу при частоте 1 МГц

Вторая проблема обусловлена тем, что в микроконтроллерах ШИМ реализуется на основе аппаратных таймеров/счетчиков, тактируемых импульсами определенной частоты *CLK*. Таймер подсчитывает количество тактовых импульсов и в нужные моменты формирует передний и задний фронты.

Если требуется получить ШИМ с частотой *F*, то количество импульсов тактовой частоты *CLK*, приходящихся на один период импульса ШИМ, определяется по формуле:

$$N = \frac{CLK}{F} \tag{5.10}$$

Чем выше *N*, тем больше мы можем задать ступеней регулирования длительности импульса ШИМ. При увеличении *F* происходит уменьшение *N*, а *CLK* нельзя поднять выше определенного уровня (для нашего микроконтроллера – 80 МГц).

Так, при *CLK* = 8 МГц и *F* = 1 МГц имеем всего 8 ступеней регулирования – совсем немного. Если опустить *F* до 100 КГц, получим уже 80 ступеней – гораздо больше.

Слишком малая частота импульсов ШИМ также приводит к определенным проблемам:

– заметное колебание мощности на нагрузке;

– переполнение аппаратного таймера/счетчика, реализующего ШИМ.

Если рассмотреть предельный случай, когда частота будет сравнима со временем реакции человека, например, 1 Гц, то будет невооруженным взглядом заметны колебания мощности на нагрузке. Лампа будет то предельно ярко загораться, то полностью гаснуть. Очевидно, что если лампа используется по прямому назначению – для освещения, то такой режим работы совершенно не приемлем. Кому понравится мерцание лампочки?

Но и при частоте 25 Гц и даже чуть выше мы сможем наблюдать мигание лампы: человеческий глаз успевает отследить его. Вообще, человеческий глаз прямым зрением (когда смотрит в упор на объект) замечает колебания до 50 Гц. Но лампа накаливания при такой частоте уже не успевает остывать, поэтому колебания становятся незаметными. Если вместо лампы накаливания взять светодиод, у которого инерционность очень мала, то колебания станут заметны. Боковым зрением (если смотреть вдоль объекта), человеческий глаз замечает мерцание с частотой до 72 Гц.

Кроме того, лампа накаливания не терпит периодических включений и выключений, если за время выключения она успевает остыть. Это приводит к быстрому перегоранию нити накала. Важно выбрать такой период импульсов, чтобы нить лампы не успевала бы значительно остывать. То же самое касается и различных теплоэлектронагревателей.

Если мы управляем скоростью вращения двигателя, то при малой частоте ШИМ будет наблюдаться прерывистое вращение вала (двигатель будет дергаться) – это тоже плохо.

Вторая проблема обусловлена тем, что аппаратные таймеры/счетчики в микроконтроллерах имеют небольшую разрядность. Так, в нашем микроконтроллере, разрядность составляет всего 16 бит. Это значит, что счетчик переполнится через 65536 импульсов. Из формулы (5.10) получим нижнюю границу частоты, при которой ШИМ еще сможет работать:

$$F_{\min} = \frac{CLK}{65536} \tag{5.11}$$

Если *CLK* = 8 МГц, то F_{min} = 122 Гц.

Можно понизить *CLK*, тогда получится достичь меньших частот (при *CLK* = 800 КГц, F_{min} = 12,3 Гц).

Перечисленные выше соображения надо всегда учитывать, разрабатывая систему с регулированием мощности с использованием ШИМ. На практике частоту ШИМ обычно выбирают порядка 10…200 КГц.

5.5. Реализация ШИМ на базе микроконтроллера

Рассмотрим, каким образом можно задействовать ШИМ в микроконтроллерах семейства 1986ВЕ9х. В таких микроконтроллерах ШИМ реализуется на базе аппаратных 16-разрядных таймеров/счетчиков TIMER1, TIMER2 и TIMER3. Каждый из них позволяет организовать до 4 каналов ШИМ (т.е. всего можно сделать 12 каналов ШИМ). Для каждого из трех таймеров можно задать свою частоту ШИМ, а для каждого канала в пределах таймера – и свою длительность импульса. Кроме того, каждый из каналов ШИМ имеет два выхода, подведенных к тому или иному выводу микроконтроллера: прямой (D) и инверсный (N). Прямой выход формирует импульсы без инверсии, а инверсный, соответственно, – с инверсией.

Соответствие выходов каналов таймеров и линий ввода-вывода микроконтроллера К1986ВЕ92QI показано в таблице 5.2. Здесь же указаны функции выводов микроконтроллера, через которые доступны каналы таймера. Напомним, что вывод микроконтроллера может иметь помимо функции линии порта ввода-вывода также основную, альтернативную, переопределенную и аналоговую функции. Также в таблице для удобства приведены номера штырей на разъемах X26 и X27 отладочной платы, на которые выведены соответствующие линии ввода-вывода.

Как видно из таблицы, не каждый канал таймера реализован. Это объясняется тем, что корпус микроконтроллера К1986ВЕ92QI содержит малое количество ножек (всего 64), поэтому не все возможности кристалла можно задействовать. В микроконтроллере 1986ВЕ91Т, имеющем корпус со 132 ножками, этого ограничения нет, и все каналы таймеров полностью реализованы. Кроме того, некоторые линии ввода-вывода, имеющиеся в микроконтроллере К1986ВЕ92QI, не выведены на разъемы X26 и X27 отладочной платы.

В нашем случае удобно использовать прямой выход канала 1 таймера TIMER1 на линии PF6 в режиме альтернативной функции, доступной на штырьке 25 разъема X27.

Таблица 5.2 – Выходы таймеров

Таймер	**Канал**	**Полярность**	**Линия порта**	**Функция**	**Штырь разъема**
TIMER1	1	D	PA1	альтернативная	12 (X27)
			PD1	основная	6 (X26)
			PF6	альтернативная	25 (X27)
		N	PA2	альтернативная	9 (X27)
			PD0	основная	5 (X26)
	2	D	PA3	альтернативная	10 (X27)
		N	PA4	альтернативная	7 (X27)
	3	D	PA5	альтернативная	8 (X27)
		N	–	–	–
	4	D	–	–	–
		N	–	–	–
TIMER2	1	D	PA1	переопределенная	12 (X27)
			PE0	альтернативная	–
		N	PA2	переопределенная	9 (X27)
			PE1	альтернативная	15 (X27)
	2	D	PA3	переопределенная	10 (X27)
		N	PA4	переопределенная	7 (X27)
	3	D	PA5	переопределенная	8 (X27)
			PE2	альтернативная	–
		N	PE3	альтернативная	16 (X27)
	4	D	–	–	–
		N	–	–	–

Продолжение таблицы 5.2

Таймер	Канал	Полярность	Линия порта	Функция	Штырь разъема
TIMER3	1	D	PB0	альтернативная	13 (X26)
			PC2	альтернативная	26 (X26)
			PD0	переопределенная	5 (X26)
			PE2	переопределенная	–
		N	PB1	альтернативная	14 (X26)
			PD1	переопределенная	6 (X26)
			PE3	переопределенная	16 (X27)
	2	D	PB2	альтернативная	15 (X26)
			PD2	переопределенная	7 (X26)
		N	PB3	альтернативная	16 (X26)
			PD3	переопределенная	8 (X26)
	3	D	PB5	переопределенная	18 (X26)
			PE6	переопределенная	–
		N	PB6	переопределенная	19 (X26)
			PE7	переопределенная	–
	4	D	PB7	переопределенная	20 (X26)
		N	PB8	переопределенная	21 (X26)

Рассмотрим проект `Lab5_1`. В заголовке `pwm.h` задаются основные параметры ШИМ:

```
// Частота импульсов ШИМ, Гц
#define PWM_PULSE_F 10000

// Коэффициент заполнения импульсов в процентах
#define PWM_PULSE_WIDTH_PERCENT 25

// Частота импульсов, подаваемых на таймер (CLK), Гц
#define PWM_TIMER_CLK 8000000
```

Смысл этих параметров был раскрыт выше.

В модуле pwm.c содержатся функции для работы с ШИМ. Настройка ШИМ осуществляется в функции U_PWM_Init. Рассмотрим это подробнее.

Для настройки нам потребуются некоторые вспомогательные переменные.

В переменной PrescalerValue будем хранить вычисленное значение предделителя тактовой частоты таймера, смысл которого будет пояснен далее:

```
uint16_t PrescalerValue;
```

В структуре TimerInitStructure типа TIMER_CntInitTypeDef разместим данные для инициализации таймера:

```
TIMER_CntInitTypeDef TimerInitStructure;
```

В структуре TimerChnInitStructure типа TIMER_ChnInitTypeDef разместим данные для инициализации канала ШИМ:

```
TIMER_ChnInitTypeDef TimerChnInitStructure;
```

В структуре TimerChnOutInitStructure типа TIMER_ChnOutInitTypeDef разместим данные для инициализации выхода канала ШИМ:

```
TIMER_ChnInitTypeDef TimerChnInitStructure;
```

Типы всех трех структур определены в заголовке MDR32F9Qx_timer.h библиотечного модуля MDR32F9Qx_timer.c, который предназначен для работы с таймерами. Естественно, этот модуль должен быть включен в состав проекта.

Разрешим тактирование таймера TIMER1 и порта PORTF:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTF | RST_CLK_PCLK_TIMER1, ENABLE);
```

Конфигурируем вывод PF6, сделав его цифровым выходом с ШИМ:

```
PORT_StructInit(&PortInitStructure);
PortInitStructure.PORT_FUNC = PORT_FUNC_ALTER;
PortInitStructure.PORT_OE = PORT_OE_OUT;
PortInitStructure.PORT_MODE = PORT_MODE_DIGITAL;
PortInitStructure.PORT_Pin = PWM_Pin;
PortInitStructure.PORT_SPEED = PORT_SPEED_MAXFAST;
PORT_Init (PWM_PORT, &PortInitStructure);
```

Немного поясним смысл этих настроек, хотя это уже изучалось нами ранее. На выводе PF6 задействуем альтернативную функцию (PORT_FUNC_ALTER), сделаем его цифровым (PORT_MODE_DIGITAL) выходом (PORT_OE_OUT) и зададим формирование как можно более крутых фронтов (PORT_SPEED_MAXFAST). Подробности о настройках линий портов изложены в разделе 2.4. При этом константа PWM_Pin, равная PORT_Pin_6, содержит номер линии ввода-вывода, а константа PWM_PORT, равная MDR_PORTF, – название порта. Обе константы определены в заголовке pwm.h.

Далее сбросим все настройки для используемого нами таймера:

```
TIMER_DeInit (PWM_TIMER);
```

Константа PWM_TIMER, равная MDR_TIMER1 и определенная в заголовке pwm.h, содержит имя используемого нами таймера.

Зададим тактирование таймера:

```
TIMER_BRGInit (PWM_TIMER, TIMER_HCLKdiv1);
```

При этом на таймер будет поступать частота, равная частоте процессорного ядра, т.е. 80 МГц. Это определяется константой TIMER_HCLKdiv1, соответствующей коэффициенту деления частоты, равному единице.

Теперь определим коэффициент деления системной тактовой частоты, чтобы получилась требуемая для нас частота тактирования таймера *CLK*. Такой коэффициент P называют предделителем частоты (prescaler). Это делается по следующей формуле:

$$P = \frac{SCLK}{CLK} - 1, \tag{5.12}$$

где *SCLK* – системная тактовая частота (в нашем случае – 80 МГц). В программе это выглядит так:

```
PrescalerValue = (uint16_t) (PWM_System_Core_Clock /
                                                PWM_TIMER_CLK) - 1;
```

При требуемой частоте *CLK* = 8 МГц получим *P* = 9. Пусть это вас не смущает: в действительности частота *SCLK* будет делиться не на *P*, а на *P* + 1, т.е. – на 10. При *P* = 0 деление не производится, т.е. *CLK* = *SCLK*.

Далее выполним конфигурирование таймера `PWM_TIMER`, заполнив структуру `TimerInitStructure`:

```
TimerInitStructure.TIMER_Prescaler = PrescalerValue;
TimerInitStructure.TIMER_Period = PWM_PULSE_PERIOD (PWM_PULSE_F);
TimerInitStructure.TIMER_CounterDirection = TIMER_CntDir_Up;
TimerInitStructure.TIMER_CounterMode = TIMER_CntMode_ClkFixedDir;
TimerInitStructure.TIMER_EventSource = TIMER_EvSrc_None;
TimerInitStructure.TIMER_FilterSampling = TIMER_FDTS_TIMER_CLK_div_1;
TimerInitStructure.TIMER_ARR_UpdateMode = TIMER_ARR_Update_Immediately;
TimerInitStructure.TIMER_ETR_FilterConf = TIMER_Filter_1FF_at_TIMER_CLK;
TimerInitStructure.TIMER_ETR_Prescaler = TIMER_ETR_Prescaler_None;
TimerInitStructure.TIMER_ETR_Polarity = TIMER_ETRPolarity_NonInverted;
TimerInitStructure.TIMER_BRK_Polarity = TIMER_BRKPolarity_NonInverted;
TIMER_CntInit (PWM_TIMER,&TimerInitStructure);
```

Параметр `TIMER_Prescaler` задает значение предделителя частоты *SCLK*, о чем мы только что говорили.

Параметр `TIMER_Period` задает период импульсов ШИМ в тактах частоты *CLK*, питающей таймер. Для удобства период вычисляется с помощью макроса `PWM_PULSE_PERIOD`, определенного в заголовке `pwm.h`. Он позволяет вычислить период импульса в тактах *CLK* в зависимости от требуемой частоты ШИМ *F*:

```
#define PWM_PULSE_PERIOD(F) ((uint16_t)((uint32_t)(PWM_TIMER_CLK) /
                                                (uint32_t)(F)) - 1)
```

Параметр `TIMER_CounterDirection` задает направление счета, в нашем случае – увеличение (вверх – `TIMER_CntDir_Up`). А можно задать и обратное направление – вниз (`TIMER_CntDir_Dn`).

Параметр TIMER_CounterMode определяет режим счета. При этом требуемое нам значение TIMER_CntMode_ClkFixedDir задает фиксированное (неменяющееся) направление счета. Т.е. всегда считаем вверх.

Параметр TIMER_EventSource определяет событие, по возникновению которого происходит увеличение счетчика таймера. Значение TIMER_EvSrc_None определяет, что специальных событий нет, и таймер будет считать каждый поступающий на него импульс частотой *SCLK*.

Назначение остальных параметров сейчас рассматривать преждевременно. Это заняло бы слишком много времени и затмило бы смысл основных настроек.

Также необходимо выполнить конфигурирование канала таймера с помощью структуры TimerChnInitStructure:

```
TIMER_ChnStructInit (&TimerChnInitStructure);
TimerChnInitStructure.TIMER_CH_Number = PWM_CHN;
TimerChnInitStructure.TIMER_CH_Mode = TIMER_CH_MODE_PWM;
TimerChnInitStructure.TIMER_CH_REF_Format = TIMER_CH_REF_Format6;
TIMER_ChnInit (PWM_TIMER, &TimerChnInitStructure);
```

Параметр TIMER_CH_Mode задает режим работы канала, в нашем случае – режим ШИМ (значение TIMER_CH_MODE_PWM). Другое возможное значение – TIMER_CH_MODE_CAPTURE – переведет канал в режим захвата. Это применяется, например, при измерении частоты импульсов, поступающих на вход микроконтроллера. Сейчас об этом говорить не будем.

Пожалуй, самым сложным для объяснения и понимания параметром при настройке канала является TIMER_CH_REF_Format. Он задает способ (формат) выработки внутреннего управляющего сигнала REF. Когда REF = 1, формируемый с помощью ШИМ сигнал активен, а когда REF = 0 – пассивен. В простейшем случае можно считать, что когда REF = 1, на выходе микроконтроллера, используемого под ШИМ, тоже будет логическая единица; когда же REF = 0, то и на выходе микроконтроллера будет логический ноль. Форматов формирования сигнала REF может быть целых восемь, но чаще всего применяется один единственный формат – TIMER_CH_REF_Format6, соответствующий привычной логике работы ШИМ. Его и выберем.

Еще нужно настроить выход канала таймера с помощью структуры TimerChnOutInitStructure:

```
TIMER_ChnOutStructInit (&TimerChnOutInitStructure);
TimerChnOutInitStructure.TIMER_CH_Number = PWM_CHN;
TimerChnOutInitStructure.TIMER_CH_DirOut_Polarity =
                                        TIMER_CHOPolarity_NonInverted;
TimerChnOutInitStructure.TIMER_CH_DirOut_Source = TIMER_CH_OutSrc_REF;
TimerChnOutInitStructure.TIMER_CH_DirOut_Mode = TIMER_CH_OutMode_Output;
TimerChnOutInitStructure.TIMER_CH_NegOut_Polarity =
                                        TIMER_CHOPolarity_NonInverted;
TimerChnOutInitStructure.TIMER_CH_NegOut_Source = TIMER_CH_OutSrc_REF;
TimerChnOutInitStructure.TIMER_CH_NegOut_Mode = TIMER_CH_OutMode_Output;
TIMER_ChnOutInit (PWM_TIMER, &TimerChnOutInitStructure);
```

Параметр TIMER_CH_Number задает номер канала таймера, для которого производится настройка выхода. Параметры TIMER_CH_DirOut_Polarity, TIMER_CH_DirOut_Source и TIMER_CH_DirOut_Mode задают настройки прямого выхода канала.

Параметр TIMER_CH_DirOut_Polarity определяет, будет ли выходной сигнал ШИМ инвертирован или нет. В нашем случае выход не будет инвертирован (TIMER_CHOPolarity_NonInverted). При желании можно было бы инвертировать выход, задав значение TIMER_CHOPolarity_Inverted.

Параметр TIMER_CH_DirOut_Source определяет источник управления выходом. В нашем случае это будет сигнал REF, о котором мы уже упомянули ранее.

Параметр TIMER_CH_DirOut_Mode определяет режим работы выхода канала таймера, то есть чем будет данный выход: выходом (TIMER_CH_OutMode_Output) или входом **(!)** (TIMER_CH_OutMode_Input). Для работы ШИМ, естественно, нужен режим выхода. Режим входа требуется в том случае, когда канал таймера используется для подсчета внешних импульсов. Например, когда нужно построить на основе микроконтроллера частотомер. Это мы будем изучать в дальнейшем.

Автор понимает, что понятие «выход в режиме входа» звучит несколько абсурдно, но иных терминов, к сожалению, не придумано.

Аналогичная совокупность параметров определена и для инверсного выхода канала: TIMER_CH_NegOut_Polarity, TIMER_CH_NegOut_Source и TIMER_CH_NegOut_Mode. В принципе, в нашем примере можно было бы не задавать параметры инверсного выхода канала. Ведь мы все равно его не используем. Но для общего развития конфигурирование выполняется.

Манипулируя значениями параметров TIMER_CH_DirOut_Polarity и TIMER_CH_NegOut_Polarity, можно превращать прямой выход канала в инверсный и, наоборот, инверсный – в прямой. Главное – не запутать самого себя. ☺

Все почти готово, поэтому разрешим работу таймера:

```
TIMER_Cmd (PWM_TIMER, ENABLE);
```

Наконец, зададим коэффициент заполнения импульсов в процентах, используя константу PWM_PULSE_WIDTH_PERCENT, которая определена в заголовке pwm.h:

```
U_PWM_Set_Pulse_Width_Percent (PWM_PULSE_WIDTH_PERCENT);
```

Функция U_PWM_Set_Pulse_Width_Percent определена в этом же модуле. Она весьма проста:

```
void U_PWM_Set_Pulse_Width_Percent (uint16_t width_percent)
{
   uint16_t width;

   // Вычислить ширину импульса,
   // взяв нужное количество % от его периода
   width = PWM_PULSE_PERIOD (PWM_PULSE_F) * width_percent / 100;
   TIMER_SetChnCompare (PWM_TIMER, PWM_CHN, width);
}
```

Внутри функции коэффициент заполнения импульсов пересчитывается в длину импульса width, которая задается для требуемого таймера и канала с помощью функции TIMER_SetChnCompare. По сути, длина импульса представляет собой количество импульсов частотой *CLK*, которое должно поступить на таймер с начала периода, чтобы формируемый импульс завершился.

Что очень важно, функцию U_PWM_Set_Pulse_Width_Percent можно вызывать в любой момент времени, тем самым меняя по ходу работы устройства скважность импульсов ШИМ.

Наконец, для наглядности выведем коэффициент заполнения импульсов на ЖКИ:

```
sprintf (message, "Width %3d%%", PWM_PULSE_WIDTH_PERCENT);
U_MLT_Put_String (message, 3);
```

Теперь ШИМ работает автоматически, не используя центральный процессор. В этом можно убедиться, подключив осциллограф к выводу PF6 (штырек 25 разъема X27) и GND.

Центральному процессору остается лишь выводить бегущую строку на ЖКИ, сообщая нам, что микроконтроллер работает.

Меняя значения констант PWM_PULSE_F, PWM_PULSE_WIDTH_PERCENT и PWM_TIMER_CLK в модуле pwm.h, можно перестраивать ШИМ на разную частоту и коэффициент заполнения. Конечно, после каждого изменения нужно перестроить проект и загрузить программу в микроконтроллер.

5.6. Пример с использованием АЦП и потенциометра для плавного изменения скважности импульсов ШИМ

Рассмотрим еще один проект, в котором используется ШИМ.

Отключите питание отладочной платы, и с помощью перемычек дополнительно подключите к ней область периферийного модуля с потенциометром согласно таблице 5.3.

Таблица 5.3 – Подключение периферийного модуля к отладочной плате

№ п/п	Цвет провода	Периферийный модуль		Отладочная плата	
		Имя штыря	Имя разъема	Имя штыря	Имя разъема
1	черный	GND2	XP2	1	X26
2	синий	средний	XP2	11	X26
3	красный	+3,3V	XP2	3	X26

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab5_2\MDK-ARM`. Проверьте правильность настроек проекта, постройте проект и загрузите программу в микроконтроллер.

Программа с помощью АЦП периодически измеряет напряжение на движке потенциометра и пропорционально ему устанавливает коэффициент заполнения импульсов ШИМ.

С помощью осциллографа можно наблюдать форму импульсов, питающих лампочку. Проект `Lab5_2`, по сути, сочетает в себе логику работы из проектов `Lab5_1` и `Lab3_3`. В модуле `adc.c` производится настройка АЦП, как это было рассмотрено в главе 3, при этом используется режим с DMA. В модуле `pwm.c` производится настройка ШИМ – точно так же, как и в проекте `Lab5_1`.

В модуле `regulator.c` реализовано измерение напряжения на потенциометре с периодичностью 100 мс и вычисление коэффициента заполнения импульсов ШИМ пропорционально измеренному напряжению. Вот как выглядит программа:

```
// Задача по реализации регулятора
__task void U_Regulator_Task_Function (void)
{
   uint32_t D;   // Результат аналого-цифрового преобразования
   uint32_t i;   // Индексная переменная
   float Pulse_Width;   // Ширина импульса
   uint32_t Pulse_Width_Percent;   // Коэффициент заполнения

   // Начальная ширина импульсов
   Pulse_Width = 0;

   while(1)
   {
      // Разрешить работу DMA с АЦП,
      // запустив цикл аналого-цифровых преобразований
      DMA_Cmd (DMA_Channel_ADC1, ENABLE);

      // Дождаться окончания преобразования
      os_evt_wait_or(EVENT_ADC_EOC, 0xFFFF);
```

```
            // Усреднить результат
            for (i = 0, D = 0; i < U_ADC_BUFFER_SIZE; i++)
               D += ADC_Buffer[i];
               D /= U_ADC_BUFFER_SIZE;

            // Коэффициент заполнения импульсов в %
            Pulse_Width_Percent = D * 100 / 4095;

            // Вывести коэффициент заполнения импульсов в % на ЖКИ
            sprintf(message , "Width %3d%", Pulse_Width_Percent);
            U_MLT_Put_String (message, 3);

            // Задать коэффициент заполнения импульсов, питающих
            лампочку
            U_PWM_Set_Pulse_Width_Percent (Pulse_Width_Percent);

            // Пауза, задающая период работы регулятора
            os_dly_wait (100);
      }
   }
```

То, что касается работы с АЦП, уже было рассмотрено ранее. Поясним лишь то, что касается непосредственно ШИМ.

Коэффициент заполнения импульсов в процентах вычисляется пропорцией: значение АЦП, умноженное на 100% и отнесенное к максимальному значению АЦП.

С помощью ранее рассмотренной в проекте `Lab5_1` функции `U_PWM_Set_Pulse_Width_Percent` устанавливаем для ШИМ найденный коэффициент заполнения импульсов `Pulse_Width_Percent`.

Все это периодически повторяем.

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 5.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. В проекте `Lab5_1` последовательно задайте значения коэффициента заполнения импульсов, питающих лампу, равными 10%, 40%, 70% и 100%. Наблюдайте за яркостью горения лампы. Теоретически рассчитайте скважность импульсов для каждого случая. С помощью осциллографа измерьте период и длительность импульсов. Вычислите коэффициент заполнения импульсов, а также скважность на основе данных осциллограммы и сравните результаты измерений с исходными данными.

Пояснение. Коэффициент заполнения задается константой `PWM_PULSE_WIDTH_PERCENT`, содержащейся в заголовке `pwm.h`. Осциллограф следует подключить зажимом к заземленной поверхности (например, разъему «ADC» отладочной платы), а его щупом касаться штыря 25 разъема X27.

2. В проекте `Lab5_1` последовательно задайте значения частоты импульсов, питающих лампу, равными 50, 100, 1000, 10000 и 1000000 Гц при постоянном коэффициенте заполнения 25%. Наблюдайте за яркостью горения лампы. С помощью осциллографа измерьте частоту импульсов, обращая внимание на их форму.

Экспериментальным путем найдите нижнюю граничную частоту, при которой реальная и заданная частоты будут совпадать. Сравните с результатами, вычисленными по формуле (5.11). Также найдите верхнюю граничную частоту, при которой не будет заметного искажения формы импульсов.

Пояснение. Частота импульсов задается константой `PWM_PULSE_F`, содержащейся в заголовке `pwm.h`. Обратите внимание, что при задании частоты 50 или 100 Гц, реальная частота импульсов не соответствует заданной. Объясните это явление. На частоте 1 МГц будет наблюдаться резкое отклонение формы импульсов от прямоугольной, а лампа будет светиться более тускло.

3. В проекте `Lab5_1` задайте коэффициент заполнения импульсов равным 10%. Экспериментальным путем найдите верхнюю граничную частоту, при которой импульсы на лампе полностью исчезнут. Объясните причину исчезновения импульсов.

4. В проекте `Lab5_1` установите частоту тактирования таймера, равной 80 КГц, а коэффициент заполнения импульсов, равным 25%. Последовательно задайте значения частоты импульсов, равными 10, 25, 50, 100 и 200 Гц. Наблюдайте за лампой: на низких частотах будет заметно мигание.

Экспериментальным путем найдите нижнюю граничную частоту, при которой мигание не будет заметно, если смотреть на лампочку прямо. Также определите нижнюю граничную частоту, при которой мигание не будет заметно и боковым зрением.

Пояснение. Частота тактирования таймера задается константой `PWM_TIMER_CLK`, содержащейся в заголовке `pwm.h`.

5. Ознакомьтесь с проектом `Lab5_2`. Подключите к отладочной плате потенциометр, как было описано в разделе 5.6. Поворачивая ручку потенциометра, наблюдайте за изменением яркости лампы. С помощью осциллографа наблюдайте за изменением длительности импульсов.

Контрольные вопросы

1. Что такое ШИМ?
2. Для чего применяют ШИМ?
3. На каком принципе основано регулирование мощности с помощью ШИМ?
4. Какими средствами реализуется ШИМ в микроконтроллерах?
5. Из каких соображений выбирают частоту импульсов при использовании ШИМ?
6. Какие основные настройки следует произвести, чтобы задействовать ШИМ?

Глава 6

Аппаратные таймеры/счетчики

Цель работы: получение навыков использования аппаратных таймеров для работы с внешними импульсами.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- цифровой осциллограф-приставка Oscill;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- драйвер для осциллографа-приставки Oscill;
- приложение Oscill для осциллографа-приставки Oscill;
- примеры кода программ.

6.1. Подготовка к работе

Отключите питание отладочной платы, если оно было подключено, и с помощью перемычки соедините два контакта отладочной платы согласно таблице 6.1.

Таблица 6.1 – Соединение контактов отладочной платы

№ п/п	Цвет провода	Отладочная плата		Отладочная плата	
		Имя штыря	Имя разъема	Имя штыря	Имя разъема
1	черный	26	X26	25	X27

На обратной стороне платы установите переключатель PC2 в положение «OFF». Это нужно для того, чтобы отключить кнопку «UP» линии PC2.

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab6_1\MDK-ARM`. Подключите к отладочной плате

программатор и питание (подробности в разделе 1.5). Проверьте правильность настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

6.2. Описание проектов

В примере `Lab6_1` измеряется частота импульсов, поступающих на вход микроконтроллера, с использованием метода измерения по частоте. На ЖКИ выводится измеренная частота, выраженная в герцах. Импульсы генерируются самим же микроконтроллером на выходе PF6 с помощью канала 1 таймера TIMER1, работающего в режиме ШИМ, и по соединительному проводу поступают на вход PC2, который подключен к каналу 1 таймера TIMER3. На базе таймера TIMER3 организовано измерение частоты импульсов. Таймер TIMER2 используется для задания периода, в течение которого проводятся измерения.

Получается, что микроконтроллер сам генерирует импульсы и сам же их принимает. Частота импульсов, генерируемых с помощью ШИМ, задается в виде константы `PWM_PULSE_F` в заголовке `pwm.h`.

В примере `Lab6_2` показано, как измерить частоту импульсов, поступающих на вход микроконтроллера, используя аппаратный таймер/счетчик и метод измерения по периоду импульсов. На ЖКИ выводится измеренная частота, выраженная в герцах.

6.3. Измерение частоты импульсов

В этой работе мы продолжаем знакомство с возможностями аппаратных таймеров/счетчиков, входящих в состав микроконтроллеров семейства 1986BE9x. На сей раз рассмотрим, каким образом можно использовать таймер при работе с внешними по отношению к микроконтроллеру импульсами. Типичнейшей задачей, требующей использования таймера в таком режиме, является задача измерения частоты импульсов сигнала, поступающего на вход микроконтроллера. По сути, из микроконтроллера нужно сделать частотомер – прибор, измеряющий частоту импульсов.

Частотомер может служить, например, для измерения частоты вращения вала электродвигателя или двигателя внутреннего сгорания. В этом

случае его называют **тахометром**. На базе частотомеров в автомобилях и иных транспортных средствах также строят **спидометры**, т.е. измерители скорости движения.

Чтобы понять принципы измерения частоты придется рассмотреть немного теории [14]. Пусть требуется измерить частоту импульсов f в диапазоне $[f_{min}, f_{max}]$ с относительной погрешностью, не превышающей ε.

При измерении частоты импульсов используют два основных подхода:

- измерение частоты по частоте (звучит как тавтология, но так принято);
- измерение частоты по периоду.

6.3.1. Измерение частоты по частоте

При измерении частоты по частоте берется некоторый фиксированный временной интервал T, в течение которого подсчитывается количество поступивших импульсов k (рисунок 6.1). Тогда частота импульсов в герцах определяется по формуле (6.1):

$$f_{\text{ч}} = \frac{k}{T} \tag{6.1}$$

Интервал T, естественно, должен быть задан в секундах.

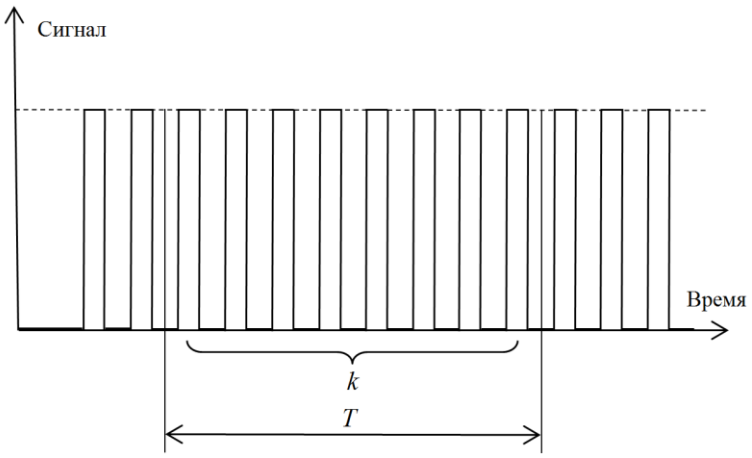


Рисунок 6.1 – Измерение частоты по частоте

Абсолютная погрешность измерения частоты по частоте может быть оценена по формуле (6.2):

$$\Delta f_{\text{ч}} = \frac{1}{T} \tag{6.2}$$

Относительная погрешность определяется так:

$$\varepsilon_{\text{ч}} = \frac{\Delta f_{\text{ч}}}{f_{\text{ч}}} = \frac{1}{k} \tag{6.3}$$

Или, используя формулу (6.1), так:

$$\varepsilon_{\text{ч}} = \frac{1}{f_{\text{ч}} \cdot T} \tag{6.4}$$

К достоинствам такого подхода можно отнести:

- низкую относительную погрешность измерений при высоких частотах, когда период импульсов гораздо меньше длины интервала T и количество поступивших импульсов k велико;
- автоматическое усреднение результатов измерений;
- простоту реализации.

Поясним достоинство, касающееся автоматического усреднения результатов измерений. Поскольку период импульсов гораздо меньше, чем интервал T, за время измерений будет рассмотрена достаточно большая выборка импульсов. Это позволит автоматически сгладить возможные неравномерности периодов импульсов. Также это позволит ослабить влияние помех (ложных импульсов), которые могут присутствовать в сигнале.

Недостатком является большая относительная погрешность измерений при низких частотах, когда период импульсов сравним с длиной интервала T и количество поступивших импульсов k мало.

Интервал времени T выбирается таким образом, чтобы с одной стороны обеспечить приемлемую точность измерений на нижней границе частоты f_{min}, а с другой стороны – не допустить переполнения счетчика импульсов на верхней границе частоты f_{max}.

Исходя из этих принципов и основываясь на формуле (6.4) можно определить, что:

$$T \geq \frac{1}{f_{min} \cdot \varepsilon_{\text{ч}}} \tag{6.5}$$

Если для подсчета импульсов используется таймер/счетчик с количеством разрядов *R*, то по формуле (6.1) получаем:

$$T \leq \frac{(2^R - 1)}{f_{max}} \tag{6.6}$$

Например, надо измерять частоту в диапазоне 1000…100000 Гц с погрешностью не хуже 1%, используя метод измерения по частоте и 16-разрядный таймер/счетчик. Нужно определить *T*.

С одной стороны, по формуле (6.5) получаем, что

$$T \geq \frac{1}{1000 \text{ Гц} \cdot \frac{1\%}{100\%}} = 0{,}1 \text{ с.}$$

С другой стороны, по формуле (6.6):

$$T \leq \frac{(2^{16} - 1)}{100000} \approx 0{,}66 \text{ с.}$$

Т.е. $T \in [0{,}1;\ 0{,}66]$ с. Для простоты реализации возьмем $T = 0{,}1$ с = 100 мс.

Также величина интервала *T* может быть ограничена сверху, если измерения нужно производить с определенной периодичностью. Допустим, если требуется производить измерения два раза в секунду, то и *T* не должен превышать 0,5 секунд.

Заметим, что не для любых исходных данных можно найти подходящее значение *T*.

6.3.2. Измерение частоты по периоду

При измерении частоты по периоду берутся вспомогательные импульсы частотой f_Z, которая должна быть гораздо больше измеряемой частоты (рисунок 6.2). Эти импульсы называют импульсами заполнения, а частоту f_Z – частотой заполнения. В течение периода импульсов измеряемой частоты $f_П$ подсчитывается количество импульсов заполнения n. Далее определяется измеряемая частота в Гц:

$$f_П = \frac{f_Z}{n} \tag{6.7}$$

Частота f_Z также должна быть задана в Гц.

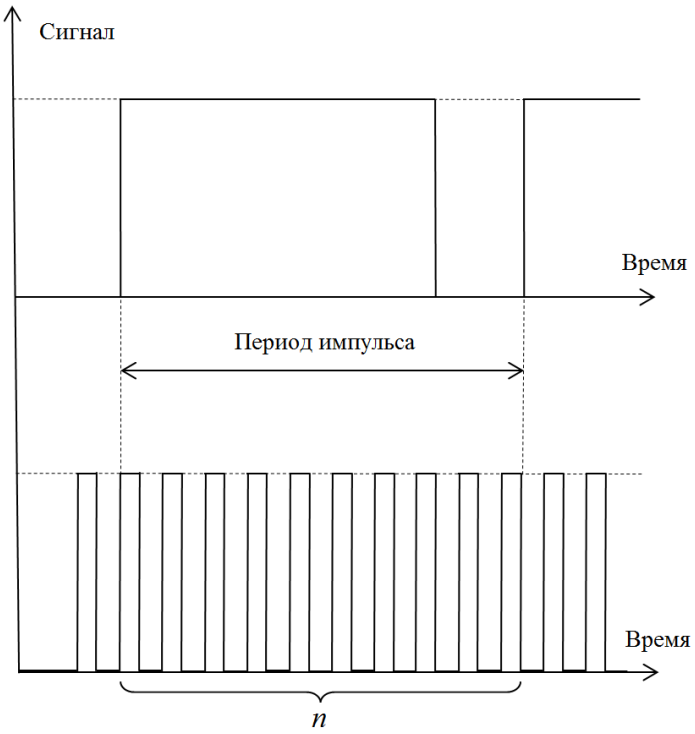


Рисунок 6.2 – Измерение частоты по периоду

Абсолютная погрешность измерения частоты по периоду может быть оценена по следующей формуле, полученной путем нахождения дифференциала от $f_П$:

$$\Delta f_П = \left|\frac{\partial f_П}{f_П} \cdot \Delta n\right| = \left|-\frac{f_Z}{n^2} \cdot 1\right| = \frac{f_Z}{n^2} \tag{6.8}$$

Относительная погрешность измерения определяется с использованием формул (6.7) и (6.8):

$$\varepsilon_ч = \frac{\Delta f_П}{f_П} = \frac{1}{n} \tag{6.9}$$

Или так:

$$\varepsilon_ч = \frac{f_П}{f_Z} \tag{6.10}$$

К достоинствам такого подхода можно отнести низкую погрешность измерений при низких измеряемых частотах, когда частота заполнения f_Z гораздо больше измеряемой частоты $f_П$ и количество поступивших импульсов заполнения n велико.

К недостаткам следует отнести:

- большую погрешность измерений при высоких измеряемых частотах, когда частота заполнения f_Z сравнима с измеряемой частотой $f_П$ и количество поступивших импульсов заполнения n мало;
- необходимость повторения измерений с последующим усреднением результатов;
- сложность реализации.

Частоту заполнения f_Z выбирают таким образом, чтобы с, одной стороны, обеспечить приемлемую точность измерений на верхней границе частоты f_{max}, а с другой стороны – не допустить переполнения счетчика импульсов на нижней границе частоты f_{min}.

Исходя из этих принципов и основываясь на формуле (6.10) можно определить:

$$f_Z \geq \frac{f_{max}}{\varepsilon} \tag{6.11}$$

Если для подсчета импульсов используется таймер/счетчик с количеством разрядов *R*, то по формуле (6.7) получаем:

$$f_Z \leq f_{min} \cdot (2^R - 1) \tag{6.12}$$

Например, надо измерять частоту в диапазоне 20…10000 Гц с погрешностью не хуже 1%, используя метод измерения по периоду и 16-разрядный таймер/счетчик. Нужно определить f_Z.

С одной стороны, по формуле (6.11) получаем:

$$f_Z \geq \frac{10000 \text{ Гц}}{\frac{1\%}{100\%}} = 1000000 \text{ Гц}.$$

С другой стороны, по формуле (6.12):

$$f_Z \leq 20 \text{ Гц} \cdot (2^{16} - 1) = 1310700 \text{ Гц}.$$

Т.е. $f_Z \in [1000000; 13100700]$ Гц. Для простоты реализации возьмем $f_Z = 10^6$ Гц = 1 МГц.

Также заметим, что не для любых исходных данных можно найти подходящее значение f_Z.

6.3.3. Усреднение результатов измерения частоты по периоду

При разговоре об измерении частоты по периоду мы упомянули недостаток, касающийся необходимости повтора измерений с последующим усреднением результатов. Поясним его суть.

Если оценить частоту, взяв лишь один период измеряемого импульса, то может получиться так (и наверняка получится!), что из-за неравномерности периодов импульсов или возникшей помехи результат измерений, являющийся случайной величиной, будет сильно отличаться от средней частоты импульсов. Поэтому, вспомнив основы математической

статистики, надо найти оценку математического ожидания $\overline{M(f_{\text{П}})}$ случайной величины $f_{\text{П}}$.

Самое простое – это рассмотреть некоторое количество импульсов N, подсчитав для каждого из них количество импульсов заполнения n_i и найдя их среднее арифметическое:

$$\overline{M(n)} = \frac{1}{N}\sum_{i=1}^{N} n_i \tag{6.13}$$

Тогда оценку математического ожидания частоты $f_{\text{П}}$ можно вычислить по формуле:

$$\overline{M(f_{\text{П}})} = \frac{f_Z}{\overline{M(n)}} \tag{6.14}$$

Недостаток такого подхода состоит в следующем. В типичной ситуации, когда помех немного и период импульсов достаточно стабилен, получится, что небольшая часть импульсов окажется очень короткими – это импульсы-помехи, большая же часть импульсов окажется практически одинаковыми по периоду. Таким образом, в массиве результатов измерений будем иметь много практически или совсем одинаковых значений, близких к математическому ожиданию, и небольшую часть значений, сильно отличающихся от математического ожидания. Если просто найти среднее арифметическое, то сильно отличающиеся значения окажут заметное влияние на результат измерений. В то же время не составляет большого труда отсеять ненужные данные.

Одним из наиболее эффективных способов является использование так называемого **медианного фильтра**. Смысл его стоит в том, что массив с результатами измерений упорядочивается по возрастанию тем или иным способом (да хотя бы методом пузырька), а затем берется элемент с номером [N / 2], т.е. серединный элемент (медиана), и используется в качестве результата. Поскольку большее количество элементов практически одинаково, серединный элемент наверняка окажется одним из них.

Количество импульсов N должно быть достаточно большим, чтобы обеспечить надежное усреднение результатов, но в то же время не слишком большим, чтобы не увеличивать без меры время измерения. Во многих случаях достаточно взять несколько десятков импульсов. Оптимальное значение N можно вычислить, оценив дисперсию случайной величины $f_{\text{П}}$, но это выходит за рамки нашей темы.

6.3.4. Одновременное измерение частоты импульсов по частоте и по периоду

В принципе, в современной аппаратуре можно организовать измерение частоты сразу двумя способами: и по частоте, и по периоду. Это позволяет значительно расширить диапазон измеряемых частот с сохранением требуемой точности измерений.

В этом случае важно определить, при какой частоте точнее получается измерение по частоте, а при какой – по периоду. Очевидно, что есть некоторая критическая частота f_{extr}, при которой относительные погрешности измерений будут одинаковыми. Ее нетрудно найти из уравнения:

$$\varepsilon_{\text{П}} = \varepsilon_{\text{Ч}} \tag{6.15}$$

Используя формулы (6.4) и (6.10), запишем уравнение (6.15) в следующем виде:

$$\frac{f_{\text{П}}}{f_Z} = \frac{1}{f_{\text{Ч}} \cdot T} \tag{6.16}$$

При этом $f_{\text{П}} = f_{\text{Ч}} = f_{extr}$, следовательно:

$$\frac{f_{extr}}{f_Z} = \frac{1}{f_{extr} \cdot T} \tag{6.17}$$

Решив уравнение (6,17), получим критическую частоту f_{extr}:

$$f_{extr} = \sqrt{\frac{f_Z}{T}} \tag{6.18}$$

При частотах выше f_{extr} лучшую точность дает измерение по частоте, а при частотах ниже f_{extr} – по периоду.

Поскольку значения f_Z и T постоянны, можно заранее рассчитать f_{extr}, не делая это по ходу выполнения измерений.

Ранее мы получали формулы (6.5), (6.6), (6.11) и (6.12) для оценки допустимых значений f_Z и T. Поскольку при больших частотах мы будем использовать измерение по частоте, а при малых – по периоду, можно для получения f_Z оставить лишь условие (6.6), а для получения T – условие (6.12). Тогда формулу (6.18) можно записать так:

$$f_{extr} = \sqrt{\frac{f_{min} \cdot (2^R - 1)}{\frac{(2^R - 1)}{f_{max}}}} = \sqrt{f_{min} \cdot f_{max}} \qquad (6.19)$$

На рисунке 6.3 показан фрагмент схемы алгоритма для измерения частоты f сразу двумя способами. При этом с f_{extr} надежней сравнивать $f_Ч$.

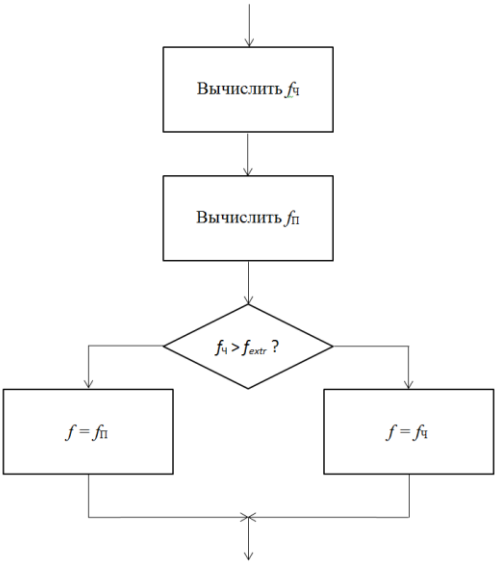


Рисунок 6.3 – Фрагмент схемы алгоритма для измерения частоты сразу двумя способами

6.4. Измерение частоты импульсов по частоте с использованием микроконтроллера

Рассмотрим особенности реализации измерения частоты по частоте на базе микроконтроллеров семейства 1986BE9x. В нашем случае – на микроконтроллере K1986BE92QI.

Рассмотрим программу проекта `Lab6_1`.

Модуль `pwm.c` реализует генерацию тестовых импульсов на выходе PF6 с помощью канала 1 таймера/счетчика TIMER1. Описание модуля `pwm.c` приведено в разделе 5.5. В заголовке `pwm.h` можно задать значение частоты генерируемых импульсов.

Генерируемые импульсы поступают по соединительному проводу на вход PC2, который подключен к каналу 1 таймера TIMER3. На базе таймера/счетчика TIMER3 организован подсчет количества измеряемых импульсов *K* в течение периода измерения *T*. Для задания периода *T* используется таймер TIMER2.

Измеренная частота выводится на ЖКИ.

Измерение частоты реализовано в модуле `fm.c`. Для повышения универсальности и наглядности программы в заголовке `fm.h` приведены некоторые константы:

```
#define FM_PORT MDR_PORTC    // Порт, в котором задействуем пин
#define FM_Pin PORT_Pin_2    // Пин для измерителя частоты
#define FM_TIMER_K MDR_TIMER3    // Таймер для подсчета импульсов K
#define FM_CHN_K TIMER_CHANNEL1    // Канал таймера
#define FM_TIMER_T MDR_TIMER2    // Таймер для отсчета периодов T
```

Таким образом, таймер TIMER3 назван константой `FM_TIMER_K`, а таймер TIMER2 – `FM_TIMER_T`. Этими именами и будем пользоваться далее.

В функции `U_FM_Task_Function` реализована задача для ОСВР RTX по измерению частоты импульсов. Задача создается привычном образом в модуле `main.c`.

```
__task void U_FM_Task_Function (void)
{
   float F; // Измеренная частота

   while(1)
   {
      // Запустить процесс измерения частоты
      FM_Start();

      // Дождаться окончания измерения
      os_evt_wait_or (EVENT_FM_READY, 0xFFFF);

      // Вычислить частоту
      F = U_FM_K / U_FM_T;

      // Вывести результат измерения частоты на ЖКИ в Гц
      sprintf(message , "F = %1.1fHz", F);
      U_MLT_Put_String (message, 3);

      // Пауза в тиках системного таймера. Здесь 1 тик = 1 мс
      os_dly_wait (250);
   }
}
```

В задаче, как всегда, организован бесконечный цикл, в котором производится запуск процесса измерения вызовом функции `FM_Start`. Когда результаты будут готовы (истечет период *T*), задача получит сообщение `EVENT_FM_READY` от обработчика прерываний таймера `FM_TIMER_T` и на основе полученных данных произведет расчет измеренной частоты:

```
F = U_FM_K / U_FM_T;
```

В глобальную переменную `U_FM_K` в обработчике прерываний от таймера `FM_TIMER_T` заносится количество подсчитанных импульсов. Константа `U_FM_T`, описанная в заголовке `fm.h`, содержит период измерения в секундах. Таким образом, имеем программную реализацию ранее рассмотренной формулы (6.1).

Полученная частота *F* выводится на ЖКИ. Через 250 мс все повторяется.

Прерывания от таймера `FM_TIMER_T` возникают по истечению периода измерения *T*. Обработчик, как обычно, реализован в модуле `MDR32F9Qx_it.c` и выглядит так:

```
// Обработчик для прерывания от TIMER2
void Timer2_IRQHandler (void)
{
   // Записать FM_TIMER_K->CNT в глобальную переменную
   U_FM_K = FM_TIMER_K->CNT;

   // Запретить работу FM_TIMER_T
   TIMER_Cmd (FM_TIMER_T, DISABLE);

   // Сбросить флаг прерывания от захвата по переполнению
   TIMER_ClearITPendingBit (FM_TIMER_T, TIMER_STATUS_CNT_ARR);

   // Установить событие для задачи измерения частоты
   isr_evt_set (EVENT_FM_READY, U_FM_Task_ID);
}
```

Здесь сначала сохраняется результат подсчета количества измеряемых импульсов из главного регистра счета таймера `FM_TIMER_K` в глобальную переменную `U_FM_K`, затем запрещается дальнейшая работа таймера `FM_TIMER_T` и сбрасываются флаги прерывания от него. Сбросить флаги очень важно, иначе мы будем бесконечно заходить в этот обработчик. Наконец, устанавливается событие для задачи по измерению частоты о том, что первичные данные для измерения готовы.

В функциях `U_FM_Init`, `TIM_K_Config` и `TIM_T_Config` выполняется инициализация и конфигурирование таймеров.

В функции `TIM_K_Config` производится конфигурирование таймера `FM_TIMER_K` для подсчета количества измеряемых импульсов *K*.

Таймер работает в режиме подсчета внешних импульсов. Результат подсчета, как уже отмечалось, считывается в глобальную переменную `U_FM_K` в подпрограмме-обработчике прерываний от таймера `FM_TIMER_T` по истечении периода измерения *T*.

Настройка таймера выглядит так. Как обычно, создаются структуры для инициализации таймера, его канала, выхода канала и линии вывода-вывода для выхода канала таймера:

```
// Структура данных для инициализации таймера
TIMER_CntInitTypeDef TimerInitStructure;
// Структура данных для инициализации канала таймера
TIMER_ChnInitTypeDef TimerChnInitStructure;
// Структура данных для инициализации выхода канала таймера
TIMER_ChnOutInitTypeDef TimerChnOutInitStructure;
// Структура данных для инициализации линий ввода-вывода
PORT_InitTypeDef PortInitStructure;
```

Разрешается тактирование таймера и порта, в котором расположена линия ввода-вывода для выхода канала таймера:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTC | RST_CLK_PCLK_TIMER3, ENABLE);
```

Линия ввода-вывода конфигурируется как цифровой **вход** (выход канала таймера будет работать в режиме входа ☺):

```
PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_FUNC = PORT_FUNC_ALTER;
PortInitStructure.PORT_OE = PORT_OE_IN;
PortInitStructure.PORT_MODE = PORT_MODE_DIGITAL;
PortInitStructure.PORT_Pin = FM_Pin;
PORT_Init (FM_PORT, &PortInitStructure);
```

При этом в данном случае задействуется альтернативная функция пина. В разделе 5.5 была приведена таблица с описанием всех возможных выходов таймеров для микроконтроллера K1986BE92QI.

Задаем тактирование таймера:

```
TIMER_BRGInit (FM_TIMER_K, TIMER_HCLKdiv1); // 80 МГц
```

Сбрасываем все настройки таймера:

```
TIMER_DeInit (FM_TIMER_K);
```

Затем заполняем структуру для инициализации таймера:

```
TIMER_CntStructInit (&TimerInitStructure);
TimerInitStructure.TIMER_Prescaler = 0;
TimerInitStructure.TIMER_Period = 65535;
TimerInitStructure.TIMER_CounterDirection = TIMER_CntDir_Up;
TimerInitStructure.TIMER_CounterMode = TIMER_CntMode_EvtFixedDir;
TimerInitStructure.TIMER_EventSource = TIMER_EvSrc_CH1;
```

```
TimerInitStructure.TIMER_ARR_UpdateMode = TIMER_ARR_Update_Immediately;
TimerInitStructure.TIMER_FilterSampling = TIMER_FDTS_TIMER_CLK_div_1;
TimerInitStructure.TIMER_ETR_FilterConf =
                                        TIMER_Filter_8FF_at_FTDS_div_32;
TimerInitStructure.TIMER_ETR_Prescaler = TIMER_ETR_Prescaler_None;
TimerInitStructure.TIMER_ETR_Polarity = TIMER_ETRPolarity_NonInverted;
TimerInitStructure.TIMER_BRK_Polarity = TIMER_BRKPolarity_NonInverted;
TIMER_CntInit (FM_TIMER_K, &TimerInitStructure);
```

Предделитель импульсов нам не нужен (будем считать все входящие импульсы), поэтому устанавливливаем значение `TIMER_IniCounter = 0`, что соответствует делению на 1. Период таймера `TIMER_Period = 65535` задаем максимально возможным, чтобы не ограничивать зря возможности таймера по подсчету импульсов высоких частот. Импульсы считаем вверх: `TIMER_CounterDirection = TIMER_CntDir_Up`. По окончании периода немедленно обновляем регистр сравнения ARR, т.е. вновь заносим туда значение периода `TIMER_Period = 65535`.

Главные настройки, которые превращают таймер в счетчик внешних импульсов, – это:

```
TimerInitStructure.TIMER_CounterMode = TIMER_CntMode_EvtFixedDir;
TimerInitStructure.TIMER_EventSource = TIMER_EvSrc_CH1;
```

Эти настройки специально приведены еще раз, чтобы вы не ошибались с ними, особенно с `TIMER_CounterMode`. Настройка `TIMER_CounterMode = TIMER_CntMode_EvtFixedDir` указывает, что таймер работает в режиме подсчета **внешних событий с фиксированным направлением счета**. По написанию значение `TIMER_CntMode_EvtFixedDir` мало отличается от значения `TIMER_CntMode_ClkFixedDir`, которое соответствует работе таймера в режиме подсчета внутренних тактовых импульсов. Если перепутать их, то внешние импульсы так и не поступят в таймер, и он будет считать совсем другое. Автор дважды «наступал на эти грабли», часами отыскивая ошибку, поэтому и привлекает ваше внимание к этому важному моменту.

Остальные настройки таймера сейчас рассматривать не будем.

Далее инициализируем требуемый канал таймера:

```
TIMER_ChnStructInit (&TimerChnInitStructure);
TimerChnInitStructure.TIMER_CH_Number = FM_CHN_K;
TimerChnInitStructure.TIMER_CH_Mode = TIMER_CH_MODE_CAPTURE;
TimerChnInitStructure.TIMER_CH_EventSource = TIMER_CH_EvSrc_PE;
TIMER_ChnInit (FM_TIMER_K, &TimerChnInitStructure);
```

В этом кодовом фрагменте ключевой момент – задать для канала режим захвата: `TIMER_CH_Mode = TIMER_CH_MODE_CAPTURE`; и событие, по которому таймер будет вести счет: `TIMER_CH_EventSource = TIMER_CH_EvSrc_PE`. В качестве события мы выбираем передний фронт импульса, поступающего в канал. Теперь по переднему фронту импульса, поступающего на соответствующий вход микроконтроллера, таймер/счетчик `FM_TIMER_K` будет увеличиваться на 1, что нам и надо для измерения частоты. При необходимости можно выбрать задний фронт: `TIMER_CH_EvSrc_NE`.

Далее инициализируем выход канала таймера:

```
TIMER_ChnOutStructInit (&TimerChnOutInitStructure);
TimerChnOutInitStructure.TIMER_CH_Number = FM_CHN_K;
TimerChnOutInitStructure.TIMER_CH_DirOut_Polarity =
                                        TIMER_CHOPolarity_NonInverted;
TimerChnOutInitStructure.TIMER_CH_DirOut_Source =
                                              TIMER_CH_OutSrc_Only_0;
TimerChnOutInitStructure.TIMER_CH_DirOut_Mode = TIMER_CH_OutMode_Input;
TIMER_ChnOutInit (FM_TIMER_K, &TimerChnOutInitStructure);
```

Здесь основное – это задать **режим входа для выхода**.

Наконец, разрешим работу таймера:

```
TIMER_Cmd (FM_TIMER_K, ENABLE);
```

С этого момента он будет считать все импульсы, приходящие на вход PC2.

Теперь рассмотрим процесс инициализации таймера `FM_TIMER_T`, реализованный в функции `TIM_T_Config`. Тут все просто, ибо этот таймер призван лишь отсчитывать периоды измерения T и по их завершению формировать аппаратное прерывание, сигнализирующее об окончании измерения.

Задаем тактирование таймера:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_TIMER2, ENABLE);
```

Деинициализируем таймер:

```
TIMER_DeInit (FM_TIMER_T);
```

Подаем на таймер импульсы для счета частотой 80 МГц:

```
TIMER_BRGInit (FM_TIMER_T, TIMER_HCLKdiv1); // 80 МГц
```

Выполняем настройку таймера с помощью здесь же описанной структуры `TimerInitStructure`:

```
TimerInitStructure.TIMER_Prescaler = (8000 - 1); // 10 КГц
TimerInitStructure.TIMER_Period = (uint16_t)(U_FM_T * 10000);
TimerInitStructure.TIMER_CounterMode = TIMER_CntMode_ClkFixedDir;
TimerInitStructure.TIMER_CounterDirection = TIMER_CntDir_Up;
TimerInitStructure.TIMER_EventSource = TIMER_EvSrc_None;
TimerInitStructure.TIMER_ARR_UpdateMode = TIMER_ARR_Update_Immediately;
TIMER_CntInit (FM_TIMER_T, &TimerInitStructure);
```

Предделитель `TIMER_Prescaler = 7999` выбираем так, чтобы получилась частота 10 КГц. Период определяем на основе константы `U_FM_T`, задающей период в секундах, с учетом только что заданной частоты тактирования таймера: `TimerInitStructure.TIMER_Period = (uint16_t)(U_FM_T * 10000)`.

Разрешим прерывания по сравнению таймера, т.е., когда он отсчитает заданное в регистре ARR (настройка `TimerInitStructure.TIMER_Period`) количество импульсов:

```
TIMER_ITConfig (FM_TIMER_T, TIMER_STATUS_CNT_ARR, ENABLE);
```

Остается задать приоритет аппаратных прерываний от таймера и разрешить эти прерывания:

```
NVIC_SetPriority (Timer2_IRQn, 0x01);
NVIC_EnableIRQ (Timer2_IRQn);
```

Запускать таймер в работу пока не будем: это будет сделано позже в задаче `U_FM_Task_Function` путем вызова функции `FM_Start`.

6.5. SVC–функции в операционной системе RTX

Функция запуска процесса измерения FM_Start – особая. Это так называемая SVC–функция. В заголовке fm.h она значится под именем FM_Start и имеет следующий прототип:

```
void __svc(1) FM_Start(void);
```

Но в модуле fm.c вы ее под именем FM_Start не найдете. Там она числится под именем __SVC_1 и выглядит так:

```
void __SVC_1 (void)
{
   __disable_irq();

   // Сбросить таймер FM_TIMER_T
   FM_TIMER_T->CNT = 0;
   // Сбросить таймер FM_TIMER_K
   FM_TIMER_K->CNT = 0;
   // Разрешить работу FM_TIMER_T
   TIMER_Cmd (FM_TIMER_T, ENABLE);

   __enable_irq();
}
```

В этой функции нужно быстро, друг за другом, сбросить счетчики таймеров FM_TIMER_T->CNT и FM_TIMER_K->CNT, а затем тут же запустить таймер FM_TIMER_T. В идеале это нужно сделать полностью одновременно, что, к сожалению, физически невозможно. Поэтому мы должны принять меры, чтобы между выполнениями перечисленных трех действий процессор ни в коем случае не отвлекся на другие задачи. Это может произойти из-за возникновения какого-либо аппаратного прерывания, в том числе и от планировщика задач RTX. Например, в неподходящий момент RTX может переключить задачу. Тогда обнуленный таймер FM_TIMER_K начнет считать импульсы, а таймер FM_TIMER_T->CNT еще не начнет отсчет периода измерения. В результате измеренная частота может существенно отличаться от реальной.

Поэтому мы запрещаем все аппаратные прерывания в начале функции FM_Start, вызывая специальную функцию RTX __disable_irq(), а в конце вновь разрешаем прерывания функцией __enable_irq().

Но проблема состоит в том, что после запуска планировщика RTX мы не имеем права просто так в любой момент разрешить или запретить аппаратные прерывания. Это сделано в целях повышения надежности ОСРВ. Если же программисту все-таки надо управлять прерываниями, то делать это можно только из специально оформленных и зарегистрированных в RTX функций – так называемых SVC-функций.

SVC-функции (Supervisor Call) – это функции программных прерываний, которые работают в так называемом привилегированном (privileged) режиме процессора семейства Cortex-M. В этих функциях разрешено обращаться к защищенной периферии, в том числе к подсистеме управления аппаратными прерываниями NVIC. SVC-функция оформляется особым образом, пример чего мы только что рассмотрели, и регистрируется в специальном ассемблерном модуле SVC_Tables.s, который нужно включить в проект. Шаблон этого файла можно взять в папке, расположенной по адресу Keil\ARM\RL\RTX\SRC\CM и относящейся к среде Keil. Вот пример такого модуля:

```
                AREA        SVC_TABLE, CODE, READONLY

                EXPORT      SVC_Count

SVC_Cnt         EQU         (SVC_End-SVC_Table)/4
SVC_Count       DCD         SVC_Cnt

; Import user SVC functions here.
                IMPORT      __SVC_1

                EXPORT      SVC_Table
SVC_Table
; Insert user SVC functions here.
; SVC 0 used by RTL Kernel.
                DCD         __SVC_1     ; user SVC function
SVC_End

                END
```

Жирным шрифтом выделены строки, которые мы включили для регистрации нашей функции __SVC_1. Можно создать сколько угодно SVC-функций, которые будут отличаться номерами. Номер 0 задействовать нельзя, так как он зарезервирован RTX.

Вызов нашей SVC-функции можно выполнять по осмысленному имени FM_Start, которое задается в заголовке. Приведем его еще раз:

```
void __svc(1) FM_Start(void);
```

Естественно, SVC-функции надо делать как можно более короткими, чтобы не препятствовать без необходимости нормальной работе планировщика задач RTX.

Подробнее о SVC-функциях можно почитать в фирменном описании RTX, которое называется «RL-ARM User's Guide» и доступно на официальном сайте Keil.

6.6. Измерение частоты импульсов по периоду с использованием микроконтроллера

Теперь рассмотрим особенности реализации измерения частоты по периоду на базе микроконтроллеров семейства 1986BE9x.

Запустите среду Keil µVision и откройте проект, расположенный по адресу Samples\Project\Lab6_2\MDK-ARM. Проверьте правильность настроек проекта, постройте проект и загрузите программу в микроконтроллер. Как и в предыдущем проекте, на линии PF6 с помощью ШИМ на таймере TIMER1 генерируются импульсы заданной частоты и по соединительному проводу поступают на вход PC2, подключенному к каналу 1 таймера TIMER3. На ЖКИ выводится частота, измеренная по периоду с помощью TIMER3.

Измерение частоты реализовано в модуле fm.c. Для повышения универсальности и наглядности программы в заголовке fm.h приведены некоторые константы:

```
#define FM_PORT MDR_PORTC      // Порт, в котором задействуем пин
#define FM_Pin PORT_Pin_2      // Пин для измерителя частоты
#define FM_TIMER MDR_TIMER3    // Таймер для подсчета импульсов К
#define FM_CHN_K TIMER_CHANNEL1      // Канал таймера
```

Таким образом, таймер TIMER3 назван константой FM_TIMER.

Как отмечалось ранее, при измерении частоты по периоду нужно выполнять целый ряд измерений, усредняя результат. Для этого FM_TIMER работает в режиме захвата внешних импульсов с использованием DMA.

Суть здесь в следующем. Таймер FM_TIMER постоянно считает вспомогательные импульсы заполнения. В данном случае частотой 1,25 МГц. По переднему фронту импульса измеряемого сигнала, пришедшего на вход PC2, в регистр CCR1 (регистр захвата по каналу 1) таймера FM_TIMER записывается текущее значение главного счетчика CNT. Тут же возникает запрос к DMA, с помощью которого значение регистра CCR1 автоматически сохраняется в очередном элементе массива TimerCapture. Это повторяется требуемое количество раз, которое задается константой U_FM_BUFFER_SIZE в заголовке fm.h.

Т.е. для каждого пришедшего на вход микроконтроллера импульса мы запоминаем количество подсчитанных импульсов заполнения, как это и требуется в методе подсчета частоты по периоду.

Когда мы получим нужное нам количество анализируемых импульсов, цикл передачи с помощью DMA завершится и возникнет прерывание от DMA. В обработчике прерываний, содержащемся в модуле MDR32F9Qx_it.c, подготавливается новый цикл передачи, запрещается дальнейшая работа DMA с таймером TIMER3 и, главное, устанавливается событие EVENT_FM_READY, сообщающее задаче U_FM_Task_Function о том, что первичные данные в ходе измерений получены.

```
void DMA_IRQHandler (void)
{
   // Подготовить к работе новый цикл измерения частоты
   DMA_InitStructure.DMA_CycleSize = U_FM_BUFFER_SIZE;
   DMA_Init (DMA_Channel_TIM3, &DMA_Channel_InitStructure);

   // Запретить дальнейшую работу канала DMA с таймером
   DMA_Cmd (DMA_Channel_TIM3, DISABLE);

   // Установить событие об окончании цикла
   // аналого-цифрового преобразования
   isr_evt_set (EVENT_FM_READY, U_FM_Task_ID);
}
```

В функции U_FM_Task_Function реализована задача для ОСВР RTX по измерению частоты импульсов по периоду. Задача запускается в модуле main.c. Задача достаточно объемна, поэтому приведем ее текст в сокращении. Полный текст можно посмотреть в исходниках проекта.

В задаче, как всегда, организован бесконечный цикл, внутри которого производится запуск цикла измерения и обработка его результатов.

Запуск производится путем разрешения работы DMA. Затем ожидается окончание цикла измерений, т.е. появление события EVENT_FM_READY. Поскольку при отсутствии входных импульсов (при нулевой частоте) цикл никогда не завершится и событие EVENT_FM_READY никогда не возникнет, для предотвращения зависания задачи ожидание ограничивается двумя секундами (параметр со значением 2000 мс при вызове функции os_evt_wait_or). Далее проверяется результат ожидания. Если функция os_evt_wait_or завершилась из-за появления события EVENT_FM_READY, мы обрабатываем результаты. Если же функция os_evt_wait_or завершилась из-за превышения времени ожидания, мы выводим на ЖКИ сообщение об отсутствии сигнала.

Обработка результатов заключается, во-первых, в преобразовании исходного массива TimerCapture в массив TimerCaptureD, содержащий количество импульсов заполнения для каждого входного импульса. Дело в том, что после захвата очередного значения счетчика импульсов заполнения, таймер продолжает считать дальше, не обнуляя себя. Поэтому нам нужно вычислить разность между соседними элементами массива TimerCapture, которая и будет количеством импульсов заполнения за период. При этом учитываем, что время от времени счетчик переполняется, и следующий элемент массива TimerCapture будет меньше предыдущего.

Во-вторых, нужно реализовать медианный фильтр. Для этого массив TimerCaptureD упорядочивается методом пузырька и находится его средний элемент. Его и берем в качестве усредненного результата измерений периода.

Ну и, наконец, по формуле (6.7) находим искомую частоту и выводим ее на ЖКИ.

```
__task void U_FM_Task_Function (void)
{
   ...
   while(1)
   {
      // Разрешить работу DMA с таймером
      DMA_Cmd (DMA_Channel_TIM3, ENABLE);

      // Дождаться окончания измерения, но не дольше 2 секунд
      wait_result = os_evt_wait_or (EVENT_FM_READY, 2000);

      // Если ожидание закончилось из-за поступления события
      if (wait_result == OS_R_EVT)
      {

         // Получить массив со значениями периодов импульсов
         for (i = 0; i < U_FM_BUFFER_SIZE - 1; i++)
         {
            D = TimerCapture[i] - TimerCapture[i - 1];
            if (D >= 0)
               TimerCaptureD[i] = D;
            else
               TimerCaptureD[i] = 65535 + D;
         }
         // Медианный фильтр
         // Упорядочить массив TimerCaptureD методом пузырька
         ...
         // Найти значение среднего элемента массива
         T = TimerCaptureD [2 + (U_FM_BUFFER_SIZE - 2) / 2];
         ...
         F = U_FM_FZ / (float)T;

         // Вывести результат измерения частоты на ЖКИ в Гц
         ...
      }

      // Ожидание завершилось из-за превышения времени
      else
      {
         // Вывести пустой результат на ЖКИ
         ...
      }
      ...
   }
}
```

Теперь рассмотрим порядок настройки таймера и DMA для работы в режиме захвата импульсов.

В функциях `U_FM_Init`, `DMA_Config` и `TIM_Config` выполняется инициализация и конфигурирование DMA и таймера.

В функции `TIM_Config` производится конфигурирование таймера `FM_TIMER`.

Настройка таймера в режиме захвата импульсов во многом похожа на настройку таймера в режим подсчета внешних импульсов, о чем было только что упомянуто. Поэтому обратим внимание лишь на отличия.

Во-первых, нужно задать тактирование таймера `FM_TIMER` импульсами заполнения. Выбор частоты заполнения подробно обосновывался ранее. В проекте она выбрана равной 1,25 МГц. Это значение легко получить, поделив частоту ядра 80 МГц на 64:

```
// Задать тактирование таймера
TIMER_BRGInit (FM_TIMER, TIMER_HCLKdiv64);   // 1.25 МГц
```

При необходимости можно задать любую другую частоту, манипулируя вторым параметром в функции `TIMER_BRGInit` и предделителем, задаваемым полем `TimerInitStructure.TIMER_Prescaler`.

При заполнении структуры настройки таймера нужно задать режим фиксированного направления счета от импульсов тактирования, а не от внешних событий:

```
TimerInitStructure.TIMER_CounterMode = TIMER_CntMode_ClkFixedDir;
```

Событие для счета должно быть пустым:

```
TimerInitStructure.TIMER_EventSource = TIMER_EvSrc_None;
```

Период таймера задаем максимально возможным:

```
TimerInitStructure.TIMER_Period = 65535;
```

Структуры настройки канала и выхода канала такие же, как и в случае с подсчетом внешних импульсов.

Далее нужно разрешить работу DMA совместно с таймером в случае захвата по каналу 1:

```
TIMER_DMACmd (FM_TIMER, TIMER_STATUS_CCR_CAP_CH1, ENABLE);
```

Разрешаем работу таймера FM_TIMER:

```
TIMER_Cmd (FM_TIMER, ENABLE);
```

И разрешаем прерывания от DMA:

```
NVIC_EnableIRQ (DMA_IRQn);
```

Но надо еще настроить и сам DMA, что делается в функции DMA_Config. Вообще-то ее вызов предшествует вызову функции TIM_Config. Но объяснять удобней в обратном порядке.

Вначале, как обычно при работе с DMA, нужно разрешить его тактирование, вместе с интерфейсами SSP1 и SSP2, а также запретить пока аппаратные прерывания. Как уже не раз говорилось в прошлых работах, это нужно для устранения аппаратной ошибки в работе DMA.

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_DMA | RST_CLK_PCLK_SSP1 |
                 RST_CLK_PCLK_SSP2, ENABLE);
NVIC->ICPR[0] = 0xFFFFFFFF;
NVIC->ICER[0] = 0xFFFFFFFF;
```

После деинициализации DMA производится заполнение структуры DMA_InitStructure для настройки DMA в целом.

```
DMA_DeInit();
DMA_StructInit (&DMA_Channel_InitStructure);
DMA_InitStructure.DMA_SourceBaseAddr = (uint32_t)(&(FM_TIMER->CCR1));
DMA_InitStructure.DMA_DestBaseAddr = (uint32_t) TimerCapture;
DMA_InitStructure.DMA_CycleSize = U_FM_BUFFER_SIZE;
DMA_InitStructure.DMA_SourceIncSize = DMA_SourceIncNo;
DMA_InitStructure.DMA_DestIncSize = DMA_DestIncHalfword;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_NumContinuous = DMA_Transfers_1;
DMA_InitStructure.DMA_SourceProtCtrl = DMA_SourcePrivileged;
DMA_InitStructure.DMA_DestProtCtrl = DMA_DestPrivileged;
DMA_InitStructure.DMA_Mode = DMA_Mode_Basic;
```

В качестве базового адреса источника берем регистр захвата по первому каналу – FM_TIMER->CCR1. В качестве базового адреса приемника – адрес массива TimerCapture, в который складываем результаты захвата. Размер цикла соответствует размеру массива TimerCapture и задается константой U_FM_BUFFER_SIZE. Адрес источника, естественно, не изменяется, а приемника – увеличивается на полслова, так как таймер 16-разрядный и элементы массива тоже. Количество повторений циклов передачи – 1. Режим работы – базовый.

Настройка канала DMA стереотипна:

```
// Задать структуру канала
DMA_Channel_InitStructure.DMA_PriCtrlData = &DMA_InitStructure;
DMA_Channel_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_Channel_InitStructure.DMA_UseBurst = DMA_BurstClear;
DMA_Channel_InitStructure.DMA_SelectDataStructure =
DMA_CTRL_DATA_PRIMARY;
DMA_Init (DMA_Channel_TIM3, &DMA_Channel_InitStructure);
```

Для устранения ошибок в работе DMA не забываем написать:

```
MDR_DMA->CHNL_REQ_MASK_CLR = 1 << DMA_Channel_TIM3;
MDR_DMA->CHNL_USEBURST_CLR = 1 << DMA_Channel_TIM3;
```

Также задаем приоритет аппаратного прерывания от DMA:

```
NVIC_SetPriority (DMA_IRQn, 1);
```

В очередной раз мы можем убедиться в полезности и мощности механизма DMA – прямого доступа к памяти.

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 6.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. В проекте `Lab6_1` измените значение периода измерения так, чтобы можно было измерять частоту в диапазоне 100…5000 Гц с относительной погрешностью не более 0,5%. Проведите необходимые расчеты. Проверьте результаты на частотах 110 и 4900 Гц, сравните с показаниями осциллографа.

2. В проекте `Lab6_2` измените настройки так, чтобы можно было измерять частоту в диапазоне 10…1000 Гц с относительной погрешностью не более 1%. Проведите необходимые расчеты. Проверьте результаты на частотах 11 и 990 Гц, сравните с показаниями осциллографа.

Контрольные вопросы

1. Какие методы измерения частоты импульсов вы знаете?
2. В каком случае следует использовать измерение частоты по периоду?
3. Для чего используется режим захвата в аппаратных таймерах/счетчиках?
4. Какие настройки следует выполнить, чтобы таймер/счетчик работал в режиме захвата?
5. Как влияет разрядность таймера/счетчика на возможности по измерению частоты импульсов?

Глава 7

Батарейный домен

Цель работы: получение навыков использования часов реального времени и регистров аварийного сохранения батарейного домена.

Оборудование:

- отладочный комплект для микроконтроллера K1986BE92QI;
- программатор-отладчик MT-Link;
- периферийный модуль;
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10 / XP;
- среда программирования Keil µVision MDK-ARM 5.20;
- драйвер программатора MT-Link;
- примеры кода программ.

7.1. Подготовка к работе

Убедитесь в наличии батереи в слоте «BATTERY 3V» рядом с ЖКИ.

Отключите питание отладочной платы, если оно было подключено, и установите перемычку «Vbat» в положение «замкнуто», т.е. наденьте ее на оба штырька. Перемычка расположена рядом с батареей. Теперь батарея с напряжением +3 В, имеющаяся на плате, будет подключена к батарейному домену микроконтроллера.

Запустите среду Keil µVision и откройте проект, расположенный по адресу `Samples\Project\Lab7_1\MDK-ARM`. Подключите к отладочной плате программатор и питание (подробности в разделе 1.5). Проверьте правильность настроек проекта (подробности в разделе 1.7.3). Постройте проект и загрузите программу в микроконтроллер (подробности в разделах 1.6 и 1.8).

7.2. Описание проектов

В примере Lab7_1 показано, как на базе часов реального времени (RTC), входящих в состав батарейного домена, можно построить таймер – устройство, подающее пользователю сигнал через определенный промежуток времени. После включения микроконтроллера запускается таймер, по срабатыванию которого на ЖКИ появляется соответствующее сообщение. Пока таймер ведет отсчет, на ЖКИ отображается количество секунд, оставшихся до срабатывания. Через три секунды после срабатывания таймер запускается снова, и все повторяется. Время срабатывания таймера в секундах задается константой U_ALARM_WAIT_TIME в заголовке alarm.h.

В примере Lab7_2 на базе RTC реализованы часы, показывающие на ЖКИ текущие время и дату. Начальное время задается в функции U_RTC_Set_Start_DateTime() модуля rtc.c.

В примере Lab7_3 на базе RTC реализованы часы, показывающие на ЖКИ текущие время и дату. В один из регистров аварийного сохранения периодически записывается текущее время. После включения питания на ЖКИ выводится время, когда произошло последнее отключение питания.

7.3. Система тактирования микроконтроллеров семейства 1986BE9x

До сих пор мы использовали микроконтроллер, не задумываясь о том, как тактируется процессорное ядро и другие устройства в составе микроконтроллера. Настало время разобраться и с этим вопросом, а затем уже перейдем к сути работы – батарейному домену.

Микроконтроллеры семейства 1986BE9x могут тактироваться от четырех различных генераторов, реализованных внутри микроконтроллера:

- HSI (High Speed Internal) – высокоскоростной внутренний RC-генератор;
- HSE (High Speed External) – высокоскоростной генератор, стабилизированный внешним кварцевым резонатором;
- LSI (Low Speed Internal) – низкоскоростной внутренний RC-генератор;
- LSE (Low Speed Internal) – низкоскоростной генератор, стабилизированный внешним кварцевым резонатором.

Далее будем для простоты использовать лишь перечисленные выше аббревиатуры, не добавляя слово генератор.

HSI вырабатывает тактовую частоту порядка 8 МГц. Он автоматически запускается при включении питания микроконтроллера, и поначалу ядро микроконтроллера тактируется именно от него. Затем можно переключиться на тактирование от других генераторов, а HSI – отключить. Достоинство HSI состоит в том, что для тактирования не требуется каких-либо дополнительных внешних элементов: все находится внутри кристалла. Однако, серьезные недостатки HSI – это низкая точность задания частоты и, главное, низкая температурная и временная стабильность частоты. При изменении температуры окружающей среды частота будет «плавать», и мы не сможем с высокой точностью измерять временные промежутки. Например, частотомер, как в прошлой работе, сделать не получится. Поэтому HSI применяют лишь в самых простых и дешевых схемах, а в серьезных проектах не используют.

HSE позволяет вырабатывать тактовую частоту от 2 до 16 МГц. При этом частота задается и стабилизируется специальным элементом – **кварцевым резонатором** (рисунок 7.1). Электронщики обычно называют его просто «кварцем». Позволим себе тоже использовать это слово.

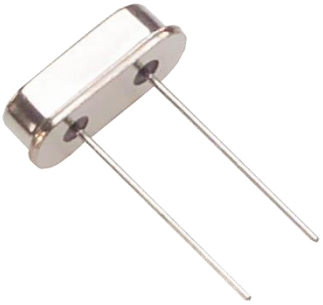


Рисунок 7.1 – Кварцевый резонатор

Основной параметр любого кварца – резонансная частота. Ее обычно пишут прямо на корпусе элемента. Например, может быть написано 8.000 МГц. При этом частота задается с точностью ±500 Гц (половина младшего разряда).

Кварц надежно стабилизирует частоту, не давая ей существенно меняться под воздействием температуры и во времени. По этой причине в мало-мальски серьезных приборах кварцы обязательно используют для тактирования процессоров. На отладочной плате вы тоже без труда найдете кварц, он расположен рядом с микроконтроллером.

На рисунке 7.2 показан фрагмент принципиальной схемы нашей отладочной платы. Как видно по схеме, тут есть сразу два кварцевых резонатора: B1 и B2. В HSE используется B1. Рядом с ним расположены маленькие (и по емкости, и по габаритам) конденсаторы C24 и C25. Они совершенно необходимы для надежного запуска генератора на кварце.

HSE можно запустить программным путем, о чем мы будем говорить далее.

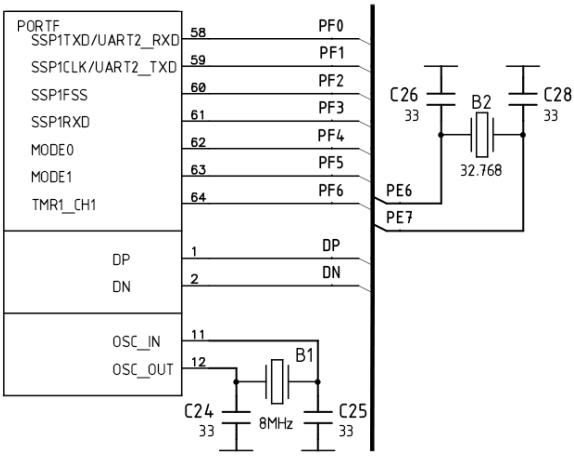


Рисунок 7.2 – Фрагменты схемы подключения кварцевых резонаторов к микроконтроллеру

LSI вырабатывает тактовую частоту порядка 40 КГц. Он автоматически запускается при включении питания. В самом начале работы LSI используется для некоторых системных нужд, а в дальнейшем его можно смело отключить программным путем. В принципе, LSI можно применить для тактирования часов реального времени, в том числе, при выключенном основном питании и работе от батареи. Но из-за низкой стабильности частоты это делается редко.

LSE предназначен для генерации частоты 32768 Гц, стабилизированной соответствующим внешним кварцем. Основное назначение LSE – тактирование часов реального времени. На схеме (рисунок 7.2) кварцевый резонатор B2 именно для этого и используется. Странное, на первый взгляд, значение частоты объясняется просто: чтобы получить частоту в 1 Гц (вести отсчет секунд), достаточно поделить 32768 Гц на 2^{15}, что сделать очень просто как аппаратными, так и программным средствами. Так и делают почти во всех электронных часах.

Итак, первый вывод таков, что в процессе нормальной работы обычно используют HSE и LSE, что мы и будем делать. Но вы, наверное, обратили внимание, что пока речь шла о сравнительно невысоких тактовых частотах – максимум 16 МГц, а на отладочной плате и того меньше – всего 8 МГц. В то же время ядро процессора может работать на частоте до 80 МГц. Откуда же ей взяться? Для этого предназначена схема формирования тактовой частоты, изображенная на рисунке 7.3. В состав схемы входят мультиплексоры (MUX), делители частоты и, главное, блоки умножения системной тактовой частоты (CPU PLL) и USB тактовой частоты (USB PLL).

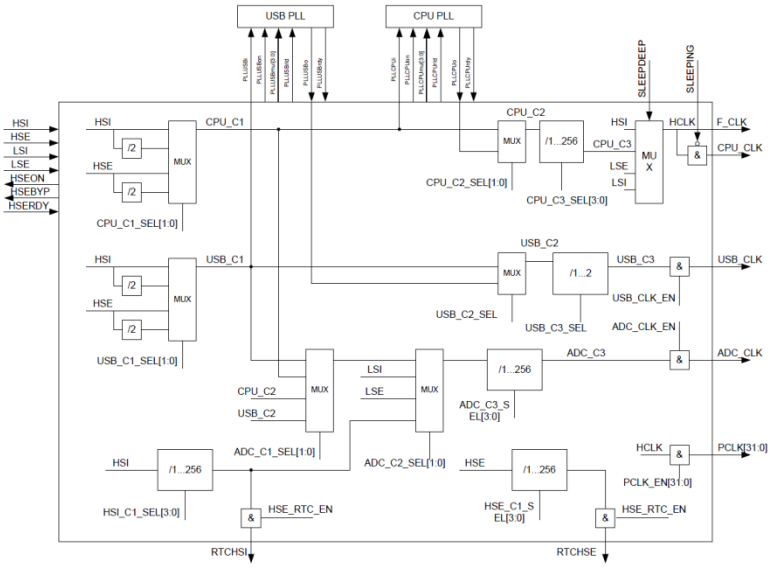


Рисунок 7.3 – Схема тактирования микроконтроллера

Встроенный блок умножения системной тактовой частоты (CPU PLL) позволяет умножить исходную частоту на коэффициент от 2 до 16, тем самым значительно повысив ее. Полученная частота и тактирует ядро микроконтроллера.

Рассмотрим конкретный пример, как нам, имея кварц 8 МГц, получить с помощью HSE частоту ядра микроконтроллера 80 МГц. В модуле `rst.c`, который практически в неизменном виде используется нами с первого же проекта, есть единственная функция `U_RST_Init()`, выполняющая настройку тактирования микроконтроллера. Эта функция всегда вызывается в самом начале программы, как только мы зайдем в функцию `main()`:

```
void U_RST_Init(void)
{
   // Включаем тактирование батарейного блока
   RST_CLK_PCLKcmd (RST_CLK_PCLK_BKP, ENABLE);

   // Включаем генератор на внешнем кварце
   RST_CLK_HSEconfig (RST_CLK_HSE_ON);
   while (RST_CLK_HSEstatus () != SUCCESS);

   // Настраиваем источник и коэффициент умножения PLL
   // (CPU_C1_SEL = HSE / 1 * 10 = 8 МГц / 1 * 10 = 80 МГц)
   RST_CLK_CPU_PLLconfig (RST_CLK_CPU_PLLsrcHSEdiv1,
                          RST_CLK_CPU_PLLmul10);

   // Включаем PLL, но еще не подключаем к кристаллу
   RST_CLK_CPU_PLLcmd (ENABLE);
   while (RST_CLK_CPU_PLLstatus () != SUCCESS);

   // Делитель C3 ( CPU_C3_SEL = CPU_C2_SEL )
   RST_CLK_CPUclkPrescaler (RST_CLK_CPUclkDIV1);

   // На C2 идет с PLL, а не напрямую с C1 (CPU_C2_SEL = PLL)
   RST_CLK_CPU_PLLuse (ENABLE);

   // CPU берет тактирование с выхода C3
   // (HCLK_SEL = CPU_C3_SEL = 80 МГц)
   RST_CLK_CPUclkSelection (RST_CLK_CPUclkCPU_C3);
}
```

Итак, после включения питания микроконтроллер начинает работать, тактируясь от HSI (частота 8 МГц). Начинаем выполнять функцию `U_RST_Init()`, вызвав ее из функции `main()`.

Если мы хотим задействовать в нашем проекте RTC с возможностью сохранения хода после отключения основного питания, первым делом нужно включить тактирование батарейного домена:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_BKP, ENABLE);
```

Заметим, что попытка вызвать в этом случае функцию деинициализации системы тактирования `RST_CLK_DeInit()` гарантировано остановит RTC во время отсутствия основного питания. Не делайте эту ошибку.

Далее нам нужно включить генератор HSE и обязательно дождаться его запуска. Генератор «разгоняется» постепенно, на это уйдет некоторое время:

```
RST_CLK_HSEconfig (RST_CLK_HSE_ON);
while (RST_CLK_HSEstatus () != SUCCESS);
```

Последующая схема процесса настройки тактирования микроконтроллера проиллюстрирована рисунком 7.4. Пунктирной ломаной линией показан тракт прохождения тактовых импульсов.

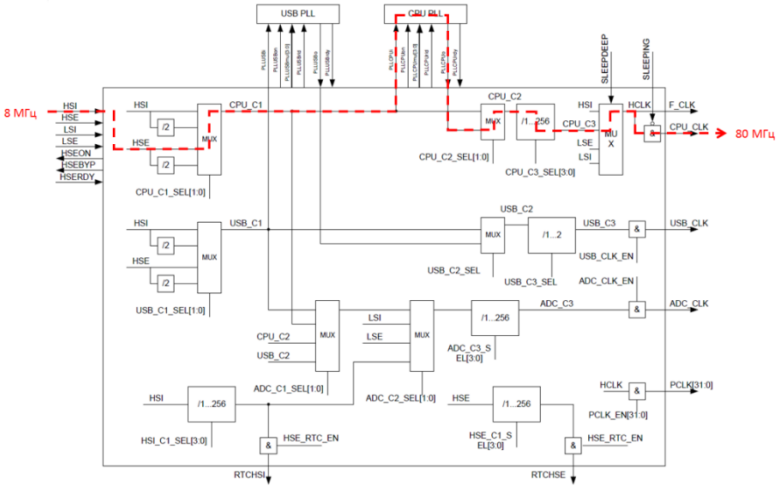


Рисунок 7.4 – Тракт прохождения тактовых импульсов

Сначала мы берем исходную частоту HSE, равную 8 МГц, и, минуя делитель на 2, подаем ее на вход мультиплексора, на выходе которого получаем частоту CPU_C1, равную тем же 8 МГц. Частота CPU_C1 подается на вход CPU PLL (умножителя частоты). Здесь мы умножаем частоту на 10 и получаем 80 МГц. Все это записывается в программе одной строчкой – вызовом специальной функции для настройки CPU PLL:

```
RST_CLK_CPU_PLLconfig (RST_CLK_CPU_PLLsrcHSEdiv1,
                       RST_CLK_CPU_PLLmul10);
```

Первый параметр позволяет выбрать источник тактирования CPU PLL: HSE/1 (как у нас), HSE/2, HSI/1 или HSI/2. Второй параметр задает коэффициент умножения частоты из предопределенного ряда: 1, 2, …16.

Теперь запустим CPU PLL и дождемся ее нормальной работы:

```
RST_CLK_CPU_PLLcmd (ENABLE);
while (RST_CLK_CPU_PLLstatus () != SUCCESS);
```

На выходе CPU PLL имеем частоту CPU_C2 = 80 МГц. Эту частоту подаем на вход мультиплексора, а с его выхода – на делитель. Мы берем коэффициент деления равный 1, т.е. по сути не делим. В результате получаем частоту CPU_C3 = 80 МГц:

```
RST_CLK_CPUclkPrescaler (RST_CLK_CPUclkDIV1);
RST_CLK_CPU_PLLuse (ENABLE);
```

А можно было бы поделить на любой коэффициент из ряда: 1, 2, 4, 8, 16, 32, 64, 128 или 256. Частота бы уменьшилась. Казалось бы, зачем это надо – понижать частоту? Оказывается, бывает нужно. Это ключевой момент в уменьшении тока потребления микроконтроллера. Ток потребления снижается почти пропорционально с понижением частоты тактирования микроконтроллера. Поэтому такая потребность может возникнуть, но не в нашем случае.

Заметим, что CPU PLL, в принципе, можно и не использовать, подав CPU_C1 сразу на делитель вместо CPU_C2. Если не требуется увеличивать частоту, то так и нужно делать. В программе мы для этого запретили бы использование CPU PLL:

```
RST_CLK_CPU_PLLuse (DISABLE);
```

Теперь частота CPU_C3 = 80 МГц через очередной мультиплексор попадет на выход схемы тактирования и пойдет к ядру микроконтроллера под именем CPU_CLK, а также к периферийным устройствам под именем HCLK или FCLK.

```
RST_CLK_CPUclkSelection (RST_CLK_CPUclkCPU_C3);
```

Именно с этого момента ядро микроконтроллера будет работать на 80 МГц, поскольку CPU_C3 заменит собой HSI, подведенное к мультиплексору, которое до сих пор использовалось по умолчанию.

Если есть необходимость, то можно вместо CPU_C3 пустить на ядро и периферию LSI или LSE, либо оставить HSI, указав в вызове функции `RST_CLK_CPUclkSelection()` соответствующее значение параметра. Но нам того не надо.

Использованные нами функции для настройки системы тактирования микроконтроллера входят в состав стандартной периферийной библиотеки и расположены в модуле `MDR32F9Qx_rst_clk.c`.

Также при настройке системы тактирования необходимо указать, какие из 32 периферийных устройств будут тактироваться. Это делается с помощью функции `RST_CLK_PCLKcmd()`, многократно встречавшейся нам в прежних проектах. В общем случае нужно задать тактирование всех устройств, которые задействованы в проекте. Но если используется DMA, то для устранения аппаратной ошибки микроконтроллеров 1986BE9x надо обязательно разрешить тактирование SSP1 и SSP2.

Например, чтобы разрешить тактирование портов PORTA и PORTD нужно написать так:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_PORTA | RST_CLK_PCLK_PORTD,
                 ENABLE);
```

Не станем здесь перечислять все возможные константы, обозначающие устройства для функции `RST_CLK_PCLKcmd()`. Просто в среде Keil подведите курсор к одной из таких констант в строке с вызовом функции `RST_CLK_PCLKcmd()`, нажмите правую кнопку мыши и выполните пункт контекстного меню «*Go To Definition Of…*». На новой вкладке откроется

заголовок MDR32F9Qx_rst_clk.h, и вы увидите полный список этих констант, обозначающих устройства.

Некоторые разработчики располагают настройку тактирования всех устройств в той же функции, где и основные настройки тактирования. По мнению автора, это нелогично. Разумнее вызывать функцию RST_CLK_PCLKcmd каждый раз в контексте настройки каждого из устройств.

Кстати, если периферийное устройство не работает, то в первую очередь надо проверить, не забыли ли мы включить его тактирование.

Следует также отметить, что не стоит задавать тактирование тех устройств, которые мы не используем. Это ведет к увеличению тока потребления микроконтроллером.

Таким образом, ничего волшебного в настройке тактирования нет. Более того, создав модуль для этой настройки один раз, вы сможете включать его во множество иных проектов.

7.4. Батарейный домен микроконтроллеров семейства 1986BE9x

В составе современных микроконтроллеров, в том числе семейства 1986BE9x, имеются специальные устройства – так называемые батарейные домены. Батарейный домен выполняет следующие основные функции:

- реализует часы реального времени (RTC – Real Time Clock), ход которых сохраняется при отключении основного питания;
- позволяет хранить небольшое количество настроек, значения которых сохраняются при отключении основного питания.

Батарейный домен может дополнительно питаться от отдельного источника питания, как правило, – батареи. При снижении напряжения основного питания микроконтроллера происходит автоматическое переключение на дополнительный источник.

Структурная схема батарейного домена приведена на рисунке 7.5.

В состав домена входят часы реального времени (RTC) и блок регистров аварийного сохранения данных. Их устройство и принцип работы рассмотрен далее.

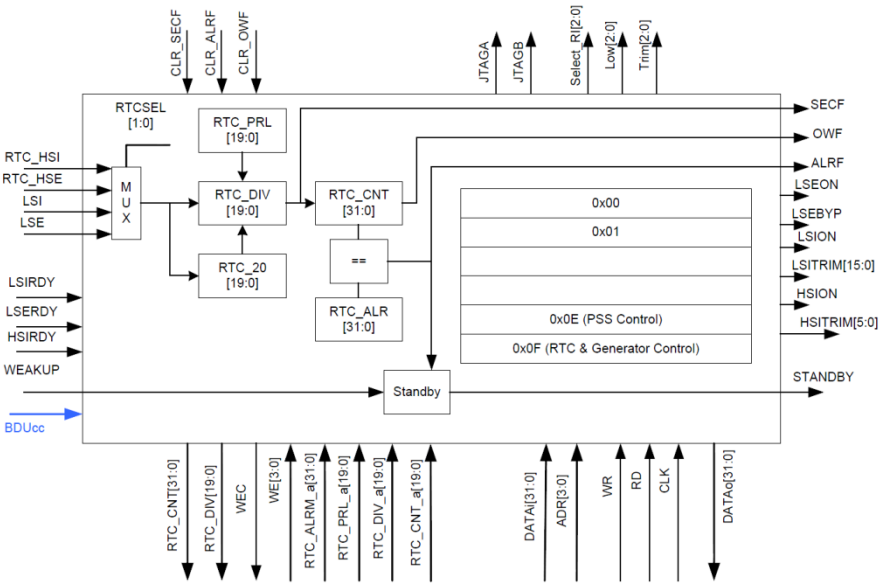


Рисунок 7.5 – Структурная схема батарейного домена

7.5. Часы реального времени

Часы реального времени обычно используют для отсчета секунд, что позволяет знать, сколько сейчас времени. Это может пригодиться, например, при ведении различных журналов, для датировки данных, отсылаемых другим устройствам, и т.п.

Кроме того, RTC позволяет отсчитывать с высокой точностью (до секунды) большие промежутки времени (минуты, часы, сутки и более). Это выгодно отличает RTC от таймеров/счетчиков общего назначения и системного таймера, используемого в RTX. Изученные нами таймеры/счетчики общего назначения хорошо справляются с интервалами времени в секунды, миллисекунды и меньше, но из-за быстрого переполнения плохо подходят для более длинных промежутков времени.

Системный таймер в RTX хорош для отсчета промежутков от миллисекунды до нескольких десятков секунд, но не больше. Даже не пытайтесь, например, отсчитать промежуток в 6 часов, используя функцию `os_dly_wait()` от ОСРВ RTX вот таким образом:

```
for (i = 0; i < 6 * 3600; i++)
{
   os_dly_wait (1000);
}
```

Отклонение по времени будет достигать нескольких минут (!). Это связано с тем, что микроконтроллер тратит время на переключение между задачами, проверку средств синхронизации и выполнение команд для организации цикла. Здесь гораздо правильней задействовать именно RTC.

RTC может тактироваться от любого из генераторов: HSE, HSI, LSE или LSI. Но чаще всего используют LSE. Это позволяет достичь высокой точности хода часов и, главное, сохранить ход часов при отключении основного питания, перейдя на питание от батареи. Собственно, из-за этого мы и говорим именно о **батарейном** домене.

Если для тактирования RTC выбраны HSE или HSI, то для понижения частоты можно использовать дополнительные делители (коэффициент деления до 256), управляемые регистрами HSE_C1_SEL или HSI_C1_SEL, в схеме тактирования микроконтроллера (рисунок 7.3). После деления на RTC поступают сигналы тактирования RTC_HSE и RTC_HSI.

Выбор источника тактирования RTC производится с помощью регистра RTCSEL (рисунок 7.5). Далее импульсы поступают на 20-разрядный делитель RTC_DIV. Коэффициент деления задается регистром RTC_PRL и может достигать 2^{20}. Значение делителя обычно задают так, чтобы на выходе получалась частота в 1 Гц.

Далее тактовые импульсы поступают на основной счетчик часов – RTC_CNT, который и будет считать секунды. В любой момент мы можем прочитать значение RTC_CNT, получив тем самым метку текущего времени. Метку времени можно затем представить в нужном нам формате. Также есть возможность задать значение RTC_CNT, выставив тем самым текущее время.

Для регулировки точности хода часов можно воспользоваться регистром RTC_20. Он показывает, какое количество тактов исходной (до деления) частоты будет пропущено из каждых 2^{20} тактов. Таким образом, мы можем при необходимости замедлить ход часов.

В составе RTC есть еще регистр RTC_ALR (от англ. alarm – тревога). В нем мы можем задать такое значение, при достижении которого счетчиком

RTC_CNT возникнет прерывание от батарейного домена. Это позволяет сделать на базе RTC таймер или будильник, позволяющий с высокой точностью измерять большие (минуты, часы, сутки и более) промежутки времени.

Кроме того, используя RTC_ALR, можно включить микроконтроллер в заданный момент времени, выведя его из режима «сна» (STANDBY). Но это отдельная тема.

7.6. Таймер на базе часов реального времени

Рассмотрим реализацию на базе RTC таймера в привычном для нормального человека понимании этого слова в проекте `Lab7_1`. Наш таймер будет через определенный промежуток времени подавать пользователю сигнал, выводя на ЖКИ соответствующее сообщение. Пока таймер ведет отсчет, на ЖКИ будет отображаться количество секунд, оставшихся до срабатывания. Через три секунды после срабатывания таймер запустится снова и все повторится.

Почти вся логика работы с RTC сосредоточена в модуле `alarm.c`. При запуске микроконтроллера из функции `main()` вызывается функция `U_Alarm_Init()`, выполняющая инициализацию батарейного домена. Выглядит это следующим образом.

Сначала разрешаем тактирование батарейного домена и порта PORTE, к двум линиям которого подключен кварц 32768 Гц:

```
RST_CLK_PCLKcmd (RST_CLK_PCLK_BKP | RST_CLK_PCLK_PORTE, ENABLE);
```

Затем инициализируем линии PE6 и PE7, к которым и подключен кварцевый резонатор (рисунок 7.2). Для этих линий задаем аналоговые функции.

```
PORT_StructInit (&PortInitStructure);
PortInitStructure.PORT_Pin = PORT_Pin_6 | PORT_Pin_7;
PortInitStructure.PORT_MODE = PORT_MODE_ANALOG;
PORT_Init (MDR_PORTE, &PortInitStructure);
```

Далее включаем генератор LSE и дожидаемся его «разгона»:

```
RST_CLK_LSEconfig (RST_CLK_LSE_ON);
while (RST_CLK_LSEstatus() != SUCCESS);
```

Выбираем LSE в качестве источника тактирования RTC, дождавшись обновления регистров RTC:

```
BKP_RTC_WaitForUpdate ();
BKP_RTCclkSource (BKP_RTC_LSEclk);
```

Заметим, что перед любым обращением к регистрам RTC нужно обязательно проверять, что их обновление закончилось. Дело в том, что операции записи в эти регистры выполняются не сразу после их вызова, а спустя некоторое время. Если вновь обратиться к RTC, не дождавшись обновления регистров, результат будет непредсказуемым. Поэтому **возьмите за правило** перед любым обращением к RTC вставлять в программу следующую строку:

```
BKP_RTC_WaitForUpdate ();
```

Теперь задаем коэффициент деления для делителя RTC_DIV, занеся в регистр RTC_PRL соответствующее значение:

```
BKP_RTC_WaitForUpdate ();
BKP_RTC_SetPrescaler (RTC_PRESCALER);
```

Здесь предделитель задан константой `RTC_PRESCALER`, определенной в заголовке `alarm.h`:

```
#define RTC_PRESCALER 32768
```

Константа равна 32768, поскольку LSE с нашей отладочной платой вырабатывает частоту 32768 Гц, а нам надо получить 1 Гц.

Далее выполняем корректировку точности хода часов, указав значение для регистра RTC_20. По сути, здесь лишь демонстрируется такая возможность, реальная же корректировка не производится, ибо на это нужно много времени (несколько суток) для проведения эксперимента. Значение константы `RTC_CALIBRATION` определено в заголовке `alarm.h` и равно нулю.

```
BKP_RTC_WaitForUpdate ();
BKP_RTC_Calibration (RTC_CALIBRATION);
```

Разрешим работу RTC:

```
BKP_RTC_WaitForUpdate ();
BKP_RTC_Enable (ENABLE);
```

Установим начальную дату и время, хотя это и не обязательно:

```
BKP_RTC_WaitForUpdate ();
U_Alarm_Set_DateTime_Stamp (RTC_INIT_TIMESTAMP);
BKP_RTC_WaitForUpdate ();
```

Наконец, разрешим прерывания от батарейного домена:

```
NVIC_SetPriority (BACKUP_IRQn, 0x02);
NVIC_EnableIRQ (BACKUP_IRQn);
```

С настройкой RTC покончено – они незаметно считают секунды.

Далее из функции `main()` привычным порядком запускается планировщик задач RTX, и создается единственная пока задача `U_Alarm_Task_Function()`, которая реализует управление нашим таймером:

```
// Задача по управлению таймером
__task void U_Alarm_Task_Function (void)
{
   time_t current;
   U_Alarm_Task_CountDown_ID = 0;
   while(1)
   {
      // Получаем текущее время
      current = U_Alarm_Get_DateTime_Stamp();

      // Создаем задачу по выводу отсчетов оставшегося времени на ЖКИ
      if (!U_Alarm_Task_CountDown_ID)
         U_Alarm_Task_CountDown_ID =
                os_tsk_create (U_Alarm_Task_CountDown_Function, 20);

      // Устанавливаем время срабатывания таймера
      BKP_RTC_WaitForUpdate ();
      BKP_RTC_SetAlarm (current + U_ALARM_WAIT_TIME);

      // Разрешаем прерывания по срабатыванию таймера RTC
      BKP_RTC_WaitForUpdate ();
      BKP_RTC_ITConfig (BKP_RTC_IT_ALRF, ENABLE);

      sprintf(message, "Waiting...");
      U_MLT_Put_String (message, 3);
```

```
        // Дожидаемся окончания измерения
        os_evt_wait_or (EVENT_ALARM, 0xFFFF);

        // Удаляем задачу по выводу отсчетов времени оставшегося
        времени на ЖКИ
        if (U_Alarm_Task_CountDown_ID)
        {
           os_tsk_delete (U_Alarm_Task_CountDown_ID);
           U_Alarm_Task_CountDown_ID = 0;
        }

        // Выводим сообщение на ЖКИ
        sprintf(message, "Alarm!");
        U_MLT_Put_String (message, 3);
        U_MLT_Put_String ("", 4);

        os_dly_wait (3000);
    }
}
```

Принцип работы задачи весьма прост. Находится текущее время и устанавливается время срабатывания таймера. При этом разрешаются прерывания по срабатыванию таймера RTC. Также временно создается еще одна задача `U_Alarm_Task_CountDown_Function()`, которая занимается выводом отсчетов времени, оставшегося до срабатывания таймера, на ЖКИ.

Далее задача останавливается в ожидании появления события о срабатывании таймера RTC. Это событие будет сформировано в обработчике прерывания от батарейного домена в модуле `MDR32F9Qx_it.c`:

```
// Обработчик для прерывания от BACKUP
void BACKUP_IRQHandler (void)
{
   // Если возникло прерывание по срабатыванию таймера RTC
   if(BKP_RTC_GetFlagStatus (BKP_RTC_FLAG_ALRF) == SET)
   {
      // Запрещаем прерывания по срабатыванию таймера RTC
      BKP_RTC_ITConfig (BKP_RTC_IT_ALRF, DISABLE);

      // Устанавливаем событие для задачи измерения частоты
      isr_evt_set (EVENT_ALARM, U_Alarm_Task_ID);
   }
}
```

Когда таймер сработает, и сообщение об этом будет получено, задача U_Alarm_Task_Function() выведет на ЖКИ сообщение о срабатывании таймера и уничтожит вспомогательную задачу U_Alarm_Task_CountDown_Function(), прекратив дальнейший вывод оставшегося до срабатывания времени на ЖКИ.

После небольшой паузы таймер будет взведен вновь, и все повторится.

Задача U_Alarm_Task_CountDown_Function() выглядит так:

```
// Задача по выводу отсчетов оставшегося времени на ЖКИ
__task void U_Alarm_Task_CountDown_Function (void)
{
   int32_t i;
   time_t current;
   time_t alarm;

   while(1)
   {
      // Получаем метку текущего времени
      current = U_Alarm_Get_DateTime_Stamp();

      // Получаем метку времени, когда таймер должен сработать
      BKP_RTC_WaitForUpdate ();
      alarm = MDR_BKP -> RTC_ALRM;

      i = alarm - current;

      // Выводим на ЖКИ количество секунд,
      // оставшихся до срабатывания таймера
      sprintf(message, "Count: %d", i);
      U_MLT_Put_String (message, 4);

      os_dly_wait (1000);
   }
}
```

Ничего сложного и интересного в ней очевидно нет, поэтому комментировать ее устройство не станем.

7.7. Электронные часы на основе RTC

Проект Lab7_2 посвящен созданию электронных часов на базе RTC. Но прежде чем реализовать их, нам нужно разобраться с метками времени.

7.7.1. Метки времени в формате UNIX Timestamp

Метки времени можно организовать по-разному. Сейчас одним из наиболее распространенных подходов является хранение меток времени в формате UNIX Timestamp или, по-другому, POSIX Timestamp. Принцип здесь очень прост. Метки времени хранятся в виде 32-битных беззнаковых целых чисел, содержащих количество секунд, прошедших с момента 00:00:00 01.01.1970 года.

Например, для момента 2 сентября 2014 г., 9:55:00 получится метка 1409651700. В Интернете есть немало сайтов для преобразования даты и времени в UNIX Timestamp и обратно.

В языке Си есть целый ряд стандартных функций для работы с датой и временем, которые расположены в модуле `time.c`. Поэтому не надо пытаться изобретать велосипед, создавая что-то подобное самостоятельно. Рассмотрим некоторые из этих функций.

Чтобы получить метку времени в формате UNIX Timestamp, имея отдельные части даты и времени, существует функция `mktime()`:

```
time_t mktime (struct tm* timeptr);
```

В качестве единственного параметра передается указатель на структуру типа `tm`, содержащей части требуемых даты и времени. В качестве результата возвращается полученная метка даты и времени. Тип `time_t` является всего-навсего переименованным в заголовке `time.h` типом `unsigned int`, т.е. 32-битным беззнаковым целым числом.

Структура `tm`, определенная в заголовке `time.h`, имеет следующее описание, приводимое с небольшими сокращениями:

```
struct tm {
   int tm_sec;     /* секунд после минуты, 0..60
   (значение 60 позволяет ввести дополнительную секунду)*/
   int tm_min;     /* минуты после часа, 0..59 */
   int tm_hour;    /* часы после полуночи, 0..23 */
   int tm_mday;    /* день месяца, 1..31 */
   int tm_mon;     /* месяцев после нала года, 0..11 (январь - 0) */
   int tm_year;    /* полных лет с 1900 года */
   int tm_wday;    /* полных дней с воскресенья, 0..6 */
   int tm_yday;    /* полных дней с 1 января, 0..365 */
};
```

Вот пример того, как можно получить метку времени для 25 мая 2012 года, 18:47:12:

```
struct tm timeinfo;
time_t TimeStamp;

// 25 мая 2012 года, 18:47:12
timeinfo.tm_sec = 12;    // Секунды (0..60)
timeinfo.tm_min = 47;    // Минуты (0..59)
timeinfo.tm_hour = 18;   // Часы (0..23)
timeinfo.tm_mday = 25;   // День месяца (1..31)
timeinfo.tm_mon = 5 - 1;       // Полных месяцев с начала года
timeinfo.tm_year = 2012 - 1900;      // Полных лет с 1900 года

TimeStamp = mktime (&timeinfo);
```

Результат получаем в переменной `TimeStamp`. Заметим, что поля `tm_wday` и `tm_yday` заполнять не надо – это избыточная информация. Эти поля пригодятся при обратном преобразовании.

Если наоборот требуется извлечь из метки времени отдельные части даты и времени, используется функция `localtime()`:

```
struct tm* localtime (const time_t* timestamp)
```

В качестве единственного параметра передается указатель на метку времени в формате UNIX Timestamp. Результатом является указатель на результирующую структуру типа `tm`, в которой и размещены все найденные части даты и времени.

Заметим, что результирующая структура является глобальной и размещена где-то в недрах модуля `tm.c`. По этой причине функция `localtime()` является нерентерабельной, т.е. ее нельзя одновременно вызывать из разных параллельно выполняющихся задач. Это надо обязательно учитывать, например, защитив доступ к ней мьютексом.

Функция `strftime()` позволяет преобразовать данные из структуры типа `tm` в строку символов, т.е. по-человечески представить дату и время.

```
size_t strftime (char* s,
                 size_t maxsize,
                 const char* format,
                 const struct tm* timeptr
                 );
```

Первый параметр `s` является указателем на начало строки, в которую записывается результат преобразования. Второй параметр `maxsize` содержит ограничение по максимальной длине формируемой результирующей строки. Третий параметр `format` содержит описание формата результирующей строки. Четвертый параметр `timeptr` является указателем на структуру данных, в которой содержатся исходные части даты и времени. В качестве результата возвращается количество символов в полученной строке или 0, если возникла ошибка.

Немного о формате результирующей строки. Функция `strftime()` подобно функции `sprintf()` распознает набор команд форматирования, начинающихся со знака «%». Команды форматирования позволяют задать точный способ представления информации о времени и дате в результирующей строке. Остальные символы, содержащиеся в строке формата, помещаются в результирующую строку без изменения. В таблице 7.1 приведен полный список команд форматирования.

Таблица 7.1 – Команды форматирования

Символ	**Значение команды**
`%a`	сокращенное наименование дня недели
`%A`	полное наименование дня недели
`%b`	сокращенное название месяца
`%B`	полное название месяца
`%c`	стандартная строка даты и времени
`%d`	день месяца в десятичном исчислении (1…31)
`%H`	час дня в пределах (0…23)

Продолжение таблицы 7.1

Символ	Значение команды
%I	час дня в пределах (1…2)
%j	день в году в десятичном представлении (1…366)
%m	месяц в десятичном представлении (1…12)
%M	минута в десятичном представлении (0…59)
%p	локальный эквивалент для AM или PM
%S	секунды в десятичном представлении (0…60)
%U	неделя в году, воскресенье служит первым днем (0…52)
%w	день недели в десятичном представлении (0…6, понедельник – 0)
%W	неделя в году, понедельник служит первым днем (0…53)
%x	стандартная строка даты
%X	стандартная строка времени
%y	год в десятичном представлении в пределах столетия (00…99)
%Y	год, включая столетия, в десятичном представлении
%Z	название временной зоны
%%	символ процента

Надо учитывать, что результат работы некоторых команд форматирования выводится на английском языке. Если же потребуется выводить названия месяцев и дней недели по-русски, то это все-таки придется реализовывать самому и уже без использования `strftime()`.

Ниже приведен пример использования функций `localtime()` и `strftime()`. Здесь реализована функция `U_RTC_Get_DateTime_String()`, которая возвращает в указанные строки `dateString` и `timeString` текущие дату и время в форматах «`dd.mm.yyyy`» и «`hh:uu:ss`»:

```
time_t U_RTC_Get_DateTime_String (char* dateString, char*
timeString)
{
   struct tm *timeinfo;
   time_t TimeStamp;

   BKP_RTC_WaitForUpdate ();
   TimeStamp = BKP_RTC_GetCounter ();

   timeinfo = localtime (&TimeStamp);
   strftime (timeString, 16, "%H:%M:%S ", timeinfo);
   strftime (dateString, 16, "%d.%m.%Y", timeinfo);

   return TimeStamp;
}
```

7.7.2. Будущие проблемы при использовании меток времени формата UNIX Timestamp

Как мы уже разобрали, метка формата UNIX Timestamp представляет собой 32-битное беззнаковое целое число, содержащее количество секунд, прошедших с момента 00:00:00 01.01.1970 года. Т.е. в такой метке можно подсчитать максимум $2^{32} – 1$ секунд. Это число достаточно велико, но не безгранично. Поэтому надо иметь в виду, если мир простоит, то рано или поздно счетчик RTC переполнится, обнулится и начнет считать заново. Это произойдет в далеком 2106 году.

В 06:28:14, 06 февраля 2106 года произойдет переполнение счетчика RTC. В следующую секунду часы покажут «01.01.1900 00:00:00» и все пойдет по новому кругу. Это проблема для наших потомков.

Но в не столь далеком 2038 году нас уже может ждать сюрприз. В 03:14:07, 19 января 2038 года, метка времени получит значение 2^{31}. Это и последующие значения могут интерпретироваться некоторыми некорректно написанными программами как отрицательные (знаковые) числа, что может привести к непредсказуемым последствиям.

В основном проблема 2038 года ждет тех, кто не пытается различать знаковые и беззнаковые целые числа и, не задумываясь, объявляет все целые переменные типом int, даже если они никогда не будут принимать отрицательные значения:

```
int i, j;                  // Возможно НЕПРАВИЛЬНО
unsigned int i, j;         // Скорее всего ПРАВИЛЬНО
```

Поэтому стоит проверять поведение наших программ в этих критических моментах, искусственно задав начальное значение RTC незадолго до указанных выше отметок времени.

7.7.3. Программная реализация электронных часов на основе RTC

В проекте `Lab7_2` на базе RTC реализованы электронные часы, показывающие на ЖКИ текущие время и дату. Начальное время задается при включении основного питания платы в функции `U_RTC_Set_Start_DateTime()` модуля `rtc.c`.

Процесс инициализации RTC практически такой же, как и в предыдущем примере, поэтому не станем его заново рассматривать. Единственное отличие состоит в том, что в конце функции `U_RTC_Init()` задается начальное время с помощью вызова функции `U_RTC_Set_Start_DateTime()`, приведенной с небольшими сокращениями:

```
void U_RTC_Set_Start_DateTime (void)
{
   struct tm timeinfo;
   time_t TimeStamp;

   // Установить начальные дату и время
   // 30.11.2014 09:55:00
   timeinfo.tm_sec = 0;         // Секунды (0..60)
   timeinfo.tm_min = 55;        // Минуты (0..59)
   timeinfo.tm_hour = 9;        // Часы (0..23)
   timeinfo.tm_mday = 30;       // День месяца (1..31)
   timeinfo.tm_mon = 11 - 1;    // Полных месяцев с начала года
                                // (0 - январь, 11 - декабрь)
   timeinfo.tm_year = 2014 - 1900;  // Полных лет с 1900 года
   ...

   // Преобразовать структуру даты и времени в отметку времени
   TimeStamp = mktime (&timeinfo);
   BKP_RTC_WaitForUpdate ();

   // Задать дату и время
   U_RTC_Set_DateTime_Stamp (TimeStamp);
}
```

В полном варианте этой функции, имеющемся в исходниках проекта, вы найдете несколько иных начальных отметок времени, заключенных в комментарии. С их помощью можно сымитировать проблемы, о которых мы говорили в предыдущем параграфе.

Единственная задача, запускаемая при старте RTX, выполняет периодический (раз в секунду) вывод текущей даты и времени на ЖКИ. Также выводится числовое значение метки времени.

```
__task void U_RTC_Task_Show_Function (void)
{
   time_t TimeStamp;

   while(1)
   {
      // Получаем текущие дату и время в формате: dd.mm.yyyy hh:uu:ss
      BKP_RTC_WaitForUpdate ();
      TimeStamp = U_RTC_Get_DateTime_String (current_date,
                                             current_time);

      // Выводим на ЖКИ текущую дату
      U_MLT_Put_String (current_date, 4);

      // Выводим на ЖКИ текущее время
      U_MLT_Put_String (current_time, 5);

      // Выводим на ЖКИ отметку времени
      sprintf (message, "%u", TimeStamp);
      U_MLT_Put_String (message, 6);

      os_dly_wait(1000);
   }
}
```

Для формирования строк с датой и временем существует описанная ниже функция `U_RTC_Get_DateTime_String()`:

```
time_t U_RTC_Get_DateTime_String (char* dateString, char*
timeString)
{
   struct tm *timeinfo;
   time_t TimeStamp;

   BKP_RTC_WaitForUpdate ();
   TimeStamp = BKP_RTC_GetCounter ();
```

```
    timeinfo = localtime (&TimeStamp);
    strftime (timeString, 16, "%H:%M:%S ", timeinfo);
    strftime (dateString, 16, "%d.%m.%Y", timeinfo);

    return TimeStamp;
}
```

7.8. Регистры аварийного сохранения

Для хранения небольшого количества произвольных настроек, значения которых будут сохраняться при отключении основного питания, в составе аварийного домена есть шестнадцать 32-разрядных регистров аварийного сохранения REG_00…REG_0F.

При этом регистры REG_0E и REG_0F используются самим батарейным доменом для сохранения настроек RTC и некоторых иных нужд. Записывать в них свои данные нельзя. Зато в остальные 14 регистров – REG_00, REG_01, REG_02, REG_03, REG_04, REG_05, REG_06, REG_07, REG_08, REG_09, REG_0A, REG_0B, REG_0C и REG_0D – можно писать все, что угодно.

Специальных функций по обращению к этим регистрам в периферийной библиотеке не предусмотрено, поэтому к ним можно обращаться просто как к полям структуры регистров батарейного домена. Например, если мы хотим записать значение 0x200100AA в регистр REG_07, то это будет выглядеть так:

```
MDR_BKP->REG_07 = 0x200100AA;
```

Аналогично, если нам надо считать значение регистра REG_0B в переменную X, то делаем так:

```
X = MDR_BKP->REG_0B;
```

По сути, с регистрами аварийного сохранения мы работаем как с обычными переменными, за исключением того, что значения переменных не исчезают после выключения питания.

Конечно, настройки прибора можно записывать и во флеш-память, но это гораздо дольше и сложней, да и количество циклов перезаписи

ограничено. Кроме того, некоторые важные данные может потребоваться срочно сохранить при внезапном исчезновении питания. В этом случае времени на работу с флеш-памятью просто не останется: счет пойдет на миллисекунды. Поэтому не стоит пренебрегать удобными и надежными регистрами аварийного сохранения.

7.9. Программная реализация аварийного сохранения данных в батарейном домене

В проекте Lab7_3 на базе RTC реализованы часы, показывающие на ЖКИ текущие время и дату. При этом в регистр REG_02 аварийного сохранения периодически записывается текущее время. После включения питания на ЖКИ выводится время, когда произошло последнее отключение питания.

В конце функции инициализации RTC после обычных настроек выполняется считывание из регистра аварийного домена времени последнего выключения основного питания платы с помощью функции U_RTC_Get_Last_OFF_DateTime(). Полученное время выводится на ЖКИ:

```
// Инициализация RTC
void U_RTC_Init (void)
{
   ...
   // Получаем дату и время последнего выключения из батарейного домена
   TimeStamp = U_RTC_Get_Last_OFF_DateTime ();

   // Выводим время последнего отключения на ЖКИ
   timeinfo = localtime (&TimeStamp);
   strftime (current_time, 16, "%H:%M:%S", timeinfo);
   U_MLT_Put_String (current_time, 3);
}
```

Функция U_RTC_Get_Last_OFF_DateTime() устроена очень просто. Здесь лишь обращение к регистру REG_02 аварийного сохранения:

```
// Получаем дату и время последнего выключения из батарейного домена
time_t U_RTC_Get_Last_OFF_DateTime (void)
{
   BKP_RTC_WaitForUpdate ();
   return MDR_BKP->REG_02;
}
```

После запуска планировщика задач RTX запускается единственная задача по выводу текущего времени и даты на ЖКИ:

```
// Задача по выводу текущего времени на ЖКИ
__task void U_RTC_Task_Show_Function (void)
{
   ...

   while(1)
   {
      ...

      // Сохраняем текущую дату и время в батарейном домене
      U_RTC_Save_DateTime ();
      // Выводим на ЖКИ текущее время
      U_MLT_Put_String (current_time, 5);
      ...

      os_dly_wait (1000);
   }
}
```

Отличие от предыдущего примера состоит лишь в том, что перед каждым выводом даты и времени на ЖКИ производится сохранение текущей метки времени в регистр аварийного сохранения REG_02 с помощью вызова функции `U_RTC_Save_DateTime()`:

```
// Сохраняем текущие дату и время в батарейном домене
void U_RTC_Save_DateTime (void)
{
   time_t TimeStamp;
   BKP_RTC_WaitForUpdate ();
   TimeStamp = BKP_RTC_GetCounter ();
   MDR_BKP->REG_02 = TimeStamp;
}
```

Таким образом, каждую секунду метка времени сохраняется в аварийных регистрах батарейного домена. Если основное питание отключится, батарейный домен будет питаться от батареи, при этом ход часов и значения аварийных регистров будут сохраняться. При возобновлении основного питания пользователь сможет узнать, когда было выключено питание платы.

Задание

Не забудьте выполнить подготовку к работе, описанную в разделе 7.1, а также резервное копирование проектов, описанное в разделе 1.1.

1. В проекте `Lab7_1` задайте частоту тактирования ядра равной 20 МГц. Убедитесь, что показания отсчета секунд стали обновляться в 4 раза медленней.

2. На основе проектов `Lab7_1` и `Lab7_2` сделайте будильник, срабатывающий в заданное в тексте программы время. На ЖКИ выводите текущее время и время срабатывания. При срабатывании будильника нужно вывести на ЖКИ сообщение «Подъем!» и зажечь лампу накаливания.

Примечание. Лампу накаливания подключите согласно таблице 5.1. ШИМ при работе с лампой использовать не надо – просто включите лампу, выдав на соответствующий вывод микроконтроллера напряжение логической единицы.

3. Используя проект `Lab7_2`, сымитируйте проблемы 2038 и 2106 годов.

4. Измените проект `Lab7_2` таким образом, чтобы месяц в текущей дате писался по-русски: ЯНВ, ФЕВ, МАР и т.д.

Контрольные вопросы

1. Для чего нужен батарейный домен?
2. Что такое часы реального времени?
3. Что требуется для сохранения работоспособности часов реального времени в случае отключения основного питания?
4. Что такое UNIX Timestamp?
5. Какие основные проблемы могут возникать при использовании RTC?
6. Каким образом можно реализовать будильник на базе RTC?
7. Как можно сохранить настройки программы в батарейном домене?
8. Как может тактироваться микроконтроллер семейства 1986BE9x?
9. В чем разница между HSE и HSI?
10. Что такое LSE?
11. Для чего нужно CPU PLL?

Рекомендации по оформлению отчетов

Формирование отчета является важной составляющей любой интеллектуальной работы. Процесс составления отчета позволяет структурировать знания, полученные в ходе выполнения работы, и создать целостное представление о предмете исследования. Преподавателю отчет необходим для оценки уровня освоения материала обучающимся и корректировки методического плана.

Из вышесказанного можно заключить, что подходить к оформлению отчетов следует столь же серьезно, как и к выполнению самой работы.

Обычно отчет оформляется на бумаге формата А4 в машинописном варианте. Допустимы рукописные правки в случае необходимости. Все листы отчета должны быть пронумерованы и соединены с помощью скобы, нити, скрепки или скоросшивателя. По договоренности с преподавателем возможно использование лишь электронной версии отчета.

В общем случае отчет должен иметь следующую структуру:

- титульный лист;
- цель работы, поставленные задачи;
- использованное оборудование;
- краткие теоретические сведения о предмете исследования;
- описание хода выполнения работы, анализ полученных результатов;
- заключение.

В раздел теоретических сведений не следует полностью копировать текст из пособия. Сведения должны фиксироваться тезисно. Рекомендуется включать в этот раздел информацию, которая оказалась новой для читателя, либо которая оценена, как важная для понимания предмета.

Особое внимание следует уделить заключению отчета, т.к. оно является, по сути, выводом проведенного научного исследования, и именно с него обычно начинают ознакомление с отчетом. Заключение представляет собой довольно краткий, формализованный текст, цель которого – дать читателю возможность получить логически ясное представление о проделанной работе и полученных результатах, не обращаясь к деталям. Написание заключения требует от автора четкого понимания, что конкретно

было сделано и получено в ходе работы, умения пользоваться научной терминологией и особого стиля изложения.

В заключениях следует использовать безличную форму предложений: «*были измерены*» вместо «*мы измерили*». Необходимо соблюдать единый стиль изложения. Не следует использовать чрезмерно усложненные грамматические конструкции и сложноподчиненные предложения или пытаться все заключение написать одной фразой. Изложение должно быть логичным и последовательным, без смысловых разрывов в тексте. [15]

Следует понимать, что отчеты в учебной работе требуются в первую очередь их авторам: они позволяют структурировать знания, полученные в ходе работы, последовательно определить и устранить области, которые были освоены недостаточно хорошо, и повторить пройденный материал. Остальные основания для формирования отчетов в учебной деятельности хоть и нельзя назвать неважными, но тем не менее являются побочными.

В приложении 2 приведен пример структуры и оформления отчета. Пример вполне может быть использован в качестве шаблона для всех отчетов в рамках работы с данной книгой, однако не является единственно допустимым вариантом. Электронную версию этого примера в формате A4 можно найти по адресу [16].

Заключение

В данной книге были рассмотрены основные аспекты программирования микроконтроллеров на базе отечественных микросхем семейства 1986BE9x разработки и производства компании «Миландр». Изложенные методы и концепции могут быть в той или иной мере экстраполированы на другие семейства микроконтроллеров (в том числе и зарубежные) для их успешного освоения.

В данную книгу не вошли разделы, связанные программированием различных интерфейсов (USB, UART, CAN, SPI, I2C): планируется издание отдельной книге на эту тему.

Автор надеется, что книга помогла читателю начать знакомство с программированием микроконтроллеров.

Читатель может без стеснения использовать предлагаемые автором примеры программ в своих как учебных, так и коммерческих проектах. Для этого они и созданы.

В завершение книги хотелось бы дать пару советов.

При освоении программирования микроконтроллеров главное – смело взяться за какой-нибудь конкретный проект. Пусть даже небольшой и не очень сложный. Хорошо, если задачу уже поставили на работе и, хочешь – не хочешь, нужно ее решить. Но нетрудно подыскать проект и самому себе. Придумайте и сделайте какое-нибудь устройство для дома, для семьи, для дачи, для автомобиля. Успешное выполнение всего одного проекта даст уверенность в собственных силах.

Постепенно осваивайте цифровую схемотехнику, берите в руки паяльник. Начать можно, например, с книг [17, 18]. Настоящий программист микроконтроллеров обязан разбираться в схемах и вообще должен быть на все руки мастером.

Список использованных источников

1. Васильев А.Е. Микроконтроллеры. Разработка встраиваемых приложений. – СПб.: БХВ-Петербург, 2008. – 304 с.: ил.

2. https://vk.com/milandrgroup (дата обращения 29.07.2016).

3. http://oscill.com (дата обращения 29.07.2016).

4. https://yadi.sk/d/kLuFKZCmtjnxr (дата обращения 29.07.2016).

5. http://www.keil.com/download/product (дата обращения 29.07.2016).

6. http://www.keil.com/mdk5/legacy (дата обращения 29.07.2016).

7. https://yadi.sk/d/dytarYWttjoib (дата обращения 29.07.2016).

8. http://www.mt-system.ru/catalog/mikrokontrolleryprocessory/nxp-semiconductors-philips/otladochnye-sredstva/mt-system-otladoc (дата обращения 29.07.2016).

9. http://milandr.ru/uploads/Products/product_80/1986%D0%92%D0%959X.pdf (дата обращения 29.07.2016).

10. https://yadi.sk/d/0mjKdQlgtjpAC (дата обращения 29.07.2016).

11. Мартин Т., Микроконтроллеры ARM7 семейств LPC2300/2400. Вводный курс разработчика / пер. с англ. А.В. Евстифеева – М.: Додэка XXI, 2010. – 336 с: ил.

12. Баранов С.И. Синтез микропрограммных автоматов (граф-схемы и автоматы). – 2-е изд., перераб. и доп.– Л.: Энергия, Ленингр. отд-ние, 1979.– 232 с., ил.

13. Оллсон, Г., Пиани Дж. Цифровые системы автоматизации и управления. – СПб.: Невский Диалект, 2001 – 557 с: ил.

14. Метрология и электрорадиоизмерения в телекоммуникационных системах. Учебное пособие / Под общей редакцией Б.Н. Тихонова. – 2-е изд., стереотип.– М.: Горячая линия – Телеком, 2012.– 360 с.: ил.

15. Анализ и представление результатов эксперимента. Учебное пособие / Под общей редакцией Вороной Н.С. – М.: НИЯУ МИФИ, 2015. – 120 с.

16. https://yadi.sk/i/NWPEhL8gtjpwB (дата обращения 29.07.2016).

17. Угрюмов Е.П. Цифровая схемотехника. – СПб.: БХВ-Санкт-Петербург, 2000. – 528 с: ил.

18. Хоровиц П., Хилл У. Искусство схемотехники. Изд. 5-е, перераб. – М.: Мир, 1998. – 698 с.

Приложение 1

В таблицах П.1 и П.2 приведена связь выводов микроконтроллера с его функциями и функциями отладочной платы.

Таблица П.1 – Линии ввода-вывода и функции микроконтроллера

Линия	Контакт	Функции			
		Основная	Альтернат.	Переопред.	Аналоговая
PA0	63	DATA0	EXT_INT1	–	–
PA1	62	DATA1	TMR1_CH1	TMR2_CH1	–
PA2	61	DATA2	TMR1_CH1N	TMR2_CH1N	–
PA3	60	DATA3	TMR1_CH2	TMR2_CH2	–
PA4	59	DATA4	TMR1_CH2N	TMR2_CH2N	–
PA5	58	DATA5	TMR1_CH3	TMR2_CH3	–
PA6	57	DATA6	CAN1_TX	UART1_RXD	–
PA7	56	DATA7	CAN1_RX	UART1_TXD	–
PB0	43	DATA16	TMR3_CH1	UART1_TXD	–
PB1	44	DATA17	TMR3_CH1N	UART2_RXD	–
PB2	45	DATA18	TMR3_CH2	CAN1_TX	–
PB3	46	DATA19	TMR3_CH2N	CAN1_RX	–
PB4	47	DATA20	TMR3_BLK	TMR3_ETR	–
PB5	50	DATA21	UART1_TXD	TMR3_CH3	–
PB6	51	DATA22	UART1_RXD	TMR3_CH3N	–
PB7	52	DATA23	nSIROUT1	TMR3_CH4	–
PB8	53	DATA24	COMP_OUT	TMR3_CH4N	–
PB9	54	DATA25	nSIRIN1	EXT_INT4	–
PB10	55	DATA26	EXT_INT2	nSIROUT1	–
PC0	42	–	SCL1	SSP2_FSS	–
PC1	41	OE	SDA1	SSP2_CLK	–
PC2	40	WE	TMR3_CH1	SSP2_RXD	–
PD0	31	TMR1_CH1N	UART2_RXD	TMR3_CH1	ADC0_R
PD1	32	TMR1_CH1	UART2_TXD	TMR3_CH1N	ADC1_R
PD2	33	BUSY1	SSP2_RXD	TMR3_CH2	ADC2
PD3	34	–	SSP2_FSS	TMR3_CH2N	ADC3
PD4	30	TMR1_ETR	nSIROUT2	TMR3_BLK	ADC4
PD5	35	CLE	SSP2_CLK	TMR2_ETR	ADC5

Продолжение таблицы П.1

Линия	Контакт	Функции микроконтроллера			
		Основная	Альтернат.	Переопред.	Аналоговая
PD6	36	ALE	SSP2_TXD	TMR2_BLK	ADC6
PD7	29	TMR1_BLK	nSIRIN2	UART1_RXD	ADC7
PE0	26	ADDR16	TMR2_CH1	CAN1_RX	DAC2_OUT
PE1	25	ADDR17	TMR2_CH1N	CAN1_TX	DAC2_REF
PE2	22	ADDR18	TMR2_CH3	TMR3_CH1	COMP_IN1
PE3	21	ADDR19	TMR2_CH3N	TMR3_CH1N	COMP_IN2
PE6	16	ADDR22	CAN2_RX	TMR3_CH3	OSC_IN32
PE7	15	ADDR23	CAN2_TX	TMR3_CH3N	OSC_OUT32
PF0	2	ADDR0	SSP1_TXD	UART2_RXD	–
PF1	3	ADDR1	SSP1_CLK	UART2_TXD	–
PF2	4	ADDR2	SSP1_FSS	CAN2_RX	–
PF3	5	ADDR3	SSP1_RXD	CAN2_TX	–
PF4	6	ADDR4	–	–	–
PF5	7	ADDR5	–	–	–
PF6	8	ADDR6	TMR1_CH1	–	–

Внимание! На линии, выделенные серым фоном, можно подавать напряжение не более 3,3 В. В случае подачи на них напряжения 5 В микроконтроллер выйдет из строя.

Таблица П.2 – Линии ввода-вывода и функции отладочной платы

Линия	Функции отладочной платы	Штыри разъемов	
		X26	X27
PA0	ЖКИ: DB0	–	11
PA1	ЖКИ: DB1	–	12
PA2	ЖКИ: DB2	–	9
PA3	ЖКИ: DB3	–	10
PA4	ЖКИ: DB4	–	7
PA5	ЖКИ: DB5	–	8
PA6	CAN: TX	–	5
PA7	CAN: RX	–	6
PB0	JTAG-A: JA_TDO	13	–
PB1	JTAG-A: JA_TMS	14	–

Продолжение таблицы П.2

Линия	Функции отладочной платы	Штыри разъемов	
		X26	X27
PB2	JTAG-A: JA_TCK	15	–
PB3	JTAG-A: JA_TDI	16	–
PB4	JTAG-A: JA_TRST	17	–
PB5	Кнопки: UP	18	–
PB6	Кнопки: RIGHT	19	–
PB7	ЖКИ: E1	20	–
PB8	ЖКИ: E2	21	–
PB9	ЖКИ: RES	22	–
PB10	ЖКИ: R/W	23	–
PC0	ЖКИ: A0; LED: 0	24	–
PC1	ЖКИ: E; LED: 1	25	–
PC2	Кнопки: SELECT	26	–
PD0	JTAG-B: JB_TDO	5	–
PD1	JTAG-B: JB_TMS	6	–
PD2	microSD: DAT0; JTAG-B: JB_TCK	7	–
PD3	microSD: CD/DAT3; JTAG-B: JB_TDI	8	–
PD4	JTAG-B: JB_TRST	9	–
PD5	microSD: CLK	10	–
PD6	microSD: CMD	11	–
PD7	Аналоговая: ADC	–	–
PE0	Аналоговая: DAC	–	–
PE1	Кнопки: DOWN	–	15
PE2	Аналоговая: CMP_IN	–	–
PE3	Кнопки: LEFT	–	16
PE6	OSC: OSC_IN32	–	–
PE7	OSC: OSC_OUT32	–	–
PF0	RS232: TX	–	19
PF1	RS232: RX	–	20
PF2	ЖКИ: DB6	–	21
PF3	ЖКИ: DB7	–	22
PF4	BOOT SELECT: MODE[0]	–	23
PF5	BOOT SELECT: MODE[1]	–	24
PF6	BOOT SELECT: MODE[2]	–	25

В таблице П.3 приведены способы вызова функций среды Keil µVision с помощью определенного сочетания клавиш.

Таблица П.3 – Горячие клавиши среды Keil µVision

Клавиши	Назначение
Сборка и отладка проекта	
F7	Построить проект
Alt + F7	Открыть окно со свойствами проекта
F5	Запустить программу или продолжить выполнение после точки останова
Ctrl + F5	Запустить / остановить отладку программы
F9	Поставить / убрать точку останова
Ctrl + Shift + F9	Удалить все точки останова
F10	Выполнить оператор без захода внутрь функции
F11	Выполнить оператор с заходом внутрь функции
Ctrl + F10	Выполнить код до выхода из функции
Ctrl + F11	Выполнить код до курсора
Форматирование и редактирование кода	
Tab	Увеличить отступ перед выделенным кодом
Shift + Tab	Уменьшить отступ перед выделенным кодом
Ctrl + Shift + U	Перевести выделенный код в верхний регистр
Ctrl + U	Перевести выделенный код в нижний регистр
Ctrl + L	Вырезать строку, на которой находиться курсор
Ctrl + Del	Удаление до конца слова
Ctrl + Backspace	Удаление до начала слова
Поиск текста	
Ctrl + F	Открыть окно поиска
Ctrl + H	Открыть окно замены
Ctrl + F3	Поиск вниз слова, на котором стоит курсор
Ctrl + Shift + F3	Поиск вверх слова, на котором стоит курсор
Ctrl + I	Быстрый поиск, текст для поиска вводится внизу главного окна
Навигация по коду	
Ctrl + G	Перейти на определённый номер строки в коде
Ctrl + Tab	Переход между открытыми вкладками
Ctrl + F2	Поставить закладку
F2	Перейти к следующей закладке
Shift + F2	Перейти к предыдущей закладке
Ctrl + Shift + F2	Убрать все закладки

Приложение 2

Пример оформления отчета по выполненной работе

Программирование микроконтроллеров

Отчет по лабораторной работе № 1

по теме

«Отладочная плата для микроконтроллера K1986BE92QI и среда программирования Keil µVision»

Выполнил: студент гр. САУ-31
Петров П.П.

Проверил: к.т.н., доцент
Иванов И.И.

Оценка: _____

« ___ » _____ 20__ г.

_____
подпись преподавателя

Москва, 2016

Отчет по лабораторной работе № 1

Отладочная плата для микроконтроллера K1986BE92QI и среда программирования Keil µVision

Цель работы: знакомство с демонстрационно-отладочной платой для микроконтроллера K1986BE92QI; получение навыков работы в среде Keil µVision; получение представления о структуре проекта на языке Си.

Оборудование: отладочный комплект для микроконтроллера K1986BE92QI; программатор-отладчик MT-Link; персональный компьютер.

Программное обеспечение: операционная система Windows 7; среда программирования Keil µVision MDK-ARM 5.20; драйвер программатора MT- Link; примеры кода программ.

Техническое задание

1) Обеспечить частоту мигания светодиодов, расположенных на отладочной плате, равной 0,2 Гц и 3 Гц соответственно.

2) Вывести на жидкокристаллический индикатор бегущую строку со словом «Байкал».

Теоретические сведения

Микроконтроллер – программируемое вычислительное устройство, обладающее набором периферийных устройств и применяемое для решения задач управления в технических системах.

Отладочная плата предназначена для ознакомления с возможностями микроконтроллера и отладки программного обеспечения для него.

Питание платы осуществляется от сети переменного тока ~220 В, 50 Гц с помощью внешнего блока питания. Также возможно питание платы от USB-интерфейса.

Для отображения буквенно-цифровой и графической информации на плате расположен **жидкокристаллический индикатор** размером 128x64.

Для загрузки программ, написанных с помощью персонального компьютера, во флеш-память микроконтроллера используется **программатор**

MT-Link. Программатор подключается к компьютеру с помощью USB-кабеля. На плате предусмотрено два разъема для подключения программатора (JTAG-A и JTAG-B).

Для **подключения отладочной платы к компьютеру** необходимо выполнить следующие действия:

1. Соединить программатор и USB-кабель между собой.
2. Подключить шлейф программатора к разъему JTAG-B, расположенному на отладочной плате.
3. Установить каждый из трех переключателей выбора режима загрузки в положение «0», что соответствует использованию интерфейса JTAG-B при загрузке и отладке.
4. Подключить USB-кабель программатора в свободный USB-порт компьютера. Убедиться, что светодиод программатора горит монотонно. Если светодиод мигает, то это значит, что нет связи между компьютером и программатором ввиду отсутствия на компьютере требуемого драйвера.
5. Установить перемычку «POWER_SEL» в положение «EXT_DC», что позволить питать плату от сети. При другом положении перемычки организуется питание со стороны USB-интерфейса.
6. Включить блок питания в сетевую розетку и подключите шнур питания к соответствующему разъему на плате. На плате должен загореться красный светодиод «POWER 5V». Если в микроконтроллере уже содержится программа, то она начнет выполняться.

Интегрированная **среда программирования** Keil µVision предназначена для написания и отладки программ для микроконтроллеров семейства ARM32 с помощью языков Си, C++ и ассемблера. В рамках нашей работы будем использовать лишь язык Си.

Программа для микроконтроллера именуется **программным проектом**. Проект представляет собой достаточно сложную совокупность файлов, каталогов и настроек.

Во-первых, проект состоит из модулей. Модуль на языке Си, как правило, состоит из двух файлов: собственно модуль – файл с расширением `*.c` и заголовочный файл с расширением `*.h`. В модуле содержатся исходные коды функций и объявления глобальных переменных,

а в заголовке – прототипы (предварительные описания) функций и глобальных переменных. Подключив заголовочный файл директивой `#include` к другому модулю, можно задействовать в одном модуле функции и переменные другого модуля.

Во-вторых, проект состоит из совокупности различных файлов, размещенных в определенной структуре каталогов на диске. В этих каталогах находятся непосредственно все файлы проекта, а также файлы библиотек, подключенных к проекту.

В-третьих, каждый проект имеет ряд настроек: программируемое устройство, тактовая частота ядра, подключаемые библиотеки, программная оптимизация, параметры программора и многие другие.

Перед загрузкой программы в микроконтроллер проект необходимо **построить**. Под построением понимается компиляция всех модулей, входящих в состав проекта, их ассемблирование и компоновку. Если в процессе построения возникнут ошибки, то сведения о них будут отображены в соответствующем окне среды. Программа не может быть загружена в микроконтроллер до тех пор, пока все ошибки не будут устранены.

Ход выполнения работы

Первая часть технического задания выполнялась на основе проекта `Lab1_1`. Проект был настроен согласно методическим рекомендациям:

- в качестве целевого устройства был установлен микроконтроллер *MDR32F9Q2I*;
- установлена стартовая тактовая частота ядра микроконтроллера равной 8 МГц;
- включено использование операционной системы RTX;
- задано использование первого уровня программной оптимизации;
- в качестве программатора выбран *J-LINK / J-TRACE Cortex* со следующими параметрами:
  - порт программирования – *SW*;
  - рабочая частота – 1 МГц;
  - стирание всей области памяти перед программированием;
  - активированы программирование, проверка и запуск программы.

Далее программный проект был построен без каких-либо ошибок и предупреждений со стороны среды Keil.

При первой попытке загрузки программы в микроконтроллер произошла ошибка, связанная с отсутствием связи программатора и среды программирования. Причиной явился неправильный выбор режима загрузки микроконтроллера с помощью переключателей на отладочной плате, вызванный нарушением инструкции по подключению отладочной платы к компьютеру.

После устранения неполадки программа была успешно загружена в память микроконтроллера.

Для выполнения задания был исследован модуль проекта `led.c` и его заголовок `led.h`.

Анализ кода модуля `led.c` позволил установить, что мигание светодиодов осуществляется с помощью задач `U_LED_Task0_Function` и `U_LED_Task1_Function`. Они имеют идентичную структуру:

```
__task void U_LED_Task0_Function (void)
{
   while(1)
   {
      U_LED_Toggle (U_LED_0_PIN);
      os_dly_wait (500);
   }
}
```

Данная задача представляет собой бесконечный цикл `while`. Внутри цикла находится функция `U_LED_Toggle`, описываемая в этом же модуле `led.c`:

```
void U_LED_Toggle (uint32_t Pins)
{
   uint32_t data = PORT_ReadInputData (U_LED_PORT);
   PORT_Write (U_LED_PORT, data ^= Pins);
}
```

Суть функции `U_LED_Toggle` состоит в следующем:

– объявляется переменная `data` типа `uint32_t` (беззнаковая, целая, 32-битная);

– переменной присваивается значение, считываемое с помощью библиотечной функции `PORT_ReadInputData` с порта `U_LED_PORT`;
– значение переменной инвертируется и записывается в порт `U_LED_PORT` с помощью библиотечной функции `PORT_Write`.

Таким образом, постоянно происходит смена состояния светодиода на противоположное ввиду бесконечного цикла.

Задержка смены состояния задается с помощью функции `os_dly_wait`. Аргументом функции служит количество тактов ядра. При данных настройках тактирования 1 такт равен 1 миллисекунде.

Из вышеизложенного очевидно, что для получения требуемых частот мигания светодиодов следует найти соответствующие им значения задержек.

Частота – это параметр периодических сигналов. Она показывает, сколько периодов такого сигнала умещается в одной секунде, и определяется формулой (1):

$$f = \frac{1}{T}, \tag{1}$$

где T – период сигнала.

Изменение управляющего сигнала светодиода можно представить графиком, изображенным на рисунке 1.

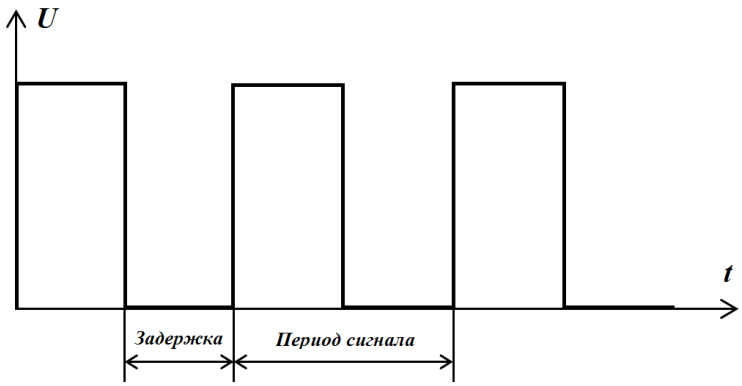


Рисунок 1 – Управляющий сигнал светодиода

Из рисунка 1 очевидно, что задержка смены состояния светодиода должна быть в два раза меньше периода управляющего сигнала.

Из формулы (1) находим требуемые значения периодов для заданных частот:

$$T_0 = \frac{1}{f_0} = \frac{1}{0{,}2\ \text{Гц}} = 5\ \text{с},$$

$$T_1 = \frac{1}{f_1} = \frac{1}{3\ \text{Гц}} \approx 0{,}33\ \text{с}.$$

Тогда искомые задержки равны 2,5 и 0,16 секунд или 2500 и 160 тактов соответственно:

```
__task void U_LED_Task0_Function (void)
{
   while(1)
   {
      U_LED_Toggle (U_LED_0_PIN);
      os_dly_wait (2500);
   }
}

__task void U_LED_Task1_Function (void)
{
   while(1)
   {
      U_LED_Toggle (U_LED_1_PIN);
      os_dly_wait (160);
   }
}
```

Наиболее достоверный способ проверки правильности полученных результатов – это измерение сигнала с помощью осциллографа. Однако использование этого прибора, видимо, выходит за рамки данной работы, поэтому единственным вариантом проверки становится визуальная оценка частоты мигания. По такой оценке **фактическая частота** вполне **соответствует требуемой.**

Вторая часть технического задания выполнялась на основе проекта Lab1_2. Данный проект, как и предыдущий, был настроен согласно методическим рекомендациям.

Для выполнения задания был исследован модуль проекта lcd.c и заголовок mlt_font.h.

Все параметры бегущей строки находятся в модуле lcd.c. Вывод бегущей строки организован в виде задачи U_LCD_Task_Function:

```
__task void U_LCD_Task_Function (void)
{
   uint32_t k = 0;

   const char s[] =
   "\xC1\xE5\xE3\xF3\xF9\xE0\xFF \xF1\xF2\xF0\xEE\xEA\xE0";

   while(1)
   {
      os_dly_wait (1000);
      U_MLT_Scroll_String (s, 4, k++);
   }
}
```

Бег строки осуществляется с помощью функции U_MLT_Scroll_String в бесконечном цикле while и основан на использовании таймера/счетчика. Данная функция имеет три аргумента:

– s – содержание строки;

– 4 – положение строки по высоте на индикаторе;

– k++ – увеличение показания счетчика на единицу.

Функция os_dly_wait в цикле фактически задает скорость бега строки.

Для выполнения технического задания необходимо изменить параметр s, определяющий содержание строки.

Латинские буквы, цифры и некоторые знаки могут записываться в содержание напрямую, однако для вывода кириллических букв нужно использовать кодировку. Такая кодировка расположена в файле mlt_font.h.

Закодированный символ представляет собой число в шестнадцатеричном формате и начинается с символов «\x». В соответствии с содержанием файла

mlt_font.h для вывода слова «Байкал» необходимо присвоить параметру s значение «\xC1\xE0\xE9\xEA\xE0\xEB»:

```
__task void U_LCD_Task_Function (void)
{
   uint32_t k = 0;

   const char s[] = "\xC1\xE0\xE9\xEA\xE0\xEB";

   while(1)
   {
      os_dly_wait (1000);
      U_MLT_Scroll_String (s, 4, k++);
   }
}
```

После загрузки программы нетрудно убедиться, что по индикатору теперь перемещается слова «Байкал».

Заключение

В ходе работы были получены базовые навыки работы с микроконтроллерами, в том числе подключение отладочной платы, запуск среды программирования, настройка, корректировка и отладка программного проекта; изучена структура программного кода для микроконтроллеров на примере двух проектов.

Единственное затруднение при работе с оборудованием было связано с неполным следованием инструкции по подключению.

Первая часть работы посвящена программному управлению частотой мигания двух светодиодов, находящихся на отладочной плате. Решением задачи стало изменение задержки при переключении состояния светодиодов на противоположное.

Вторая часть заключалась в выводе бегущей строки на жидкокристаллический индикатор. Для решения поставленной задачи потребовалось изменить аргумент функции, отвечающей за вывод строки, с использованием файла с кодировкой кириллических символов.

Результаты, полученные в ходе работы, полностью соответствуют техническому заданию.