Лабораторные работы по Архитектурам ЭВМ. Контроллер AVR.

Я

March 9, 2016

Contents

1 Общая информация о контроллерах AVR **2**

2 Toolchain **3**

3 Структура программы **7**

4 Организация памяти **11**

5 Инструкции процессора AVR **15**
5.1 Арифметические действия 15
5.2 Битовые операции 18
5.3 Инструкции перехода 19
5.4 Инструкции передачи данных 22
5.5 Используем objdump 23

6 Самостоятельная работа: моргаем светодиодами **25**

7 Таймеры и прервыания **27**

8 Самостоятельная работа: моргаем светодиодами правильно **40**

9 Последовательный интерфейс **42**

10 Самостоятельная работа: очень дорогой термометр **50**

1 Общая информация о контроллерах AVR

AVR - семейство 8-битных[1] микроконтроллеров фирмы Atmel. С момента разработки[2] было выпущено много представителей этого семейства с различными характеристиками (память, частота, форм-фактор), поэтому для определенности мы будем использовать контроллер Atmel AVR Atmega328p в DIP корпусе [Wikipedia, c]. В качестве языка программирования мы будем использовать язык ассемблера, это потому, что цель курса - знакомство с архитектурами вычислительных систем, а не достижение высот в проектировании ПО для встроенных систем, но вообще нет никаких проблем с программированием, например, на языке C.

Итак, AVR - это RISC процессор[3]. Это означает, что в наборе инструкций присутствует только небольшое [4] количество действительно необходимых команд. Более того, команды, зачастую, имеют одинаковую вычислительную сложность и длину.

В процессорах AVR есть 32 8-битных[5] регистра, которые именуются r0-r31. Стоит отметить, что не все регистры равнозначны, т. е. некоторые команды налагают ограничения на допустимые в качестве операндов регистры, но о них мы будем упоминать по мере необходимости[6].

Кроме регистров доступна и другая память расположенная прямо на чипе. Есть несколько видов памяти:

- flash - основная постоянная энергонезависимая[7] память процессоров AVR, именно здесь хранится программа, которую вы "зашиваете" в контроллер;

- eeprom - вспомогательная энергонезависимая память, в ней обычно хранят какие-либо настройки, которые могут изменяться в процессе работы программы, и которые нужно сохранять при перезапусках;

[1] по началу это может стать серьезной проблемой

[2] 1996 год

[3] есть предположение, что AVR расшифровывается как Advanced Virtual RISC, впрочем звучит сомнительно, особенно часть про Advanced

[4] относительно

[5] что логично, для 8-битных процессоров

[6] это не какая-то особенность AVR, такие ограничения есть везде, более того в RISC процессорах их гораздо меньше чем в CISC

[7] на английском это называется non-volatile

- sram - энергозависимая память AVR, которая хранит данные программы, которые будут потеряны при перезапуске (например, стек, сюда же отображаются порты ввода/вывода)

Т. е. программа для AVR сохраняется в flash памяти, во время работы она оперирует данными в sram, и по необходимости сохраняет всякие настройки в eeprom. Кстати, стоит отметить, что AVR являются представителями гарвардской архитектуры [Wikipedia, a], т. е. программа и данные разделены в памяти физически, для работы с памятью, в которой хранится программа, существуют специальные инструкции. Впрочем, мы не будем работать с программной памятью, так что для нас это малополезная информация.

Главные источники информации при работе с AVR это Datasheet [AVR, a] и описание набора инструкций процессора [AVR, b]. Эти ресурсы являются истиной в последней инстанции, и любые разногласия между данным текстом и этими документами решаются в пользу последних, так что полезно иметь их под рукой.

2 Toolchain

Toolchain - это набор инструментов и библиотек, который используется для компиляции, компоновки и загрузки на устройство программ. Первое, что необходимо сделать при знакомстве с новой технологией (будь то язык программирования, компилятор, фреймворк или какая-то библиотека), так это настроить рабочее окружение и проверить его работоспособность (на простом примере, например, hello world). Поэтому мы уделим немного времени краткому описанию утилит, которые необходимо установить для работы.

Все инструкции будут даваться для Ubuntu Linux, но версии всех пакетов существуют и для Windows[1], так что ярые фанаты последней могут адаптировать эти инструкции под себя.

Для нормальной работы требуется установить следующие пакеты:

```
1 #!/bin/bash
2
3 apt-get install avra gcc-avr binutils-avr avrdude
```

По порядку, что и зачем нужно:

- avra - ассемблер, он будет собирать из исходных файлов на языке

[1] слухи - лично не проверял

ассемблера hex файлы (не суть важно что это такое, можете считать, для простоты, что это исполняемый файл)[1];

- gcc-avr - компилятор C/C++[2], мы не будем писать на C или C++, но дальше мы увидим, зачем они могут быть полезны;
- binutils-avr - в этот пакет входит много полезных команд для работы с бинарными файлами, например, avr-objdump;
- avrdude - используя эту утилиту, мы будем загружать готовую программу на контроллер.

Возьмем за основу простейшую программу, на данный момент вам не нужно понимать, что именно и как она делает, мы просто хотим проверить работоспособность toolchain-a:

```
1  #define bit(x) (1 << (x))
2  #define DDRB   0x04
3  #define PORTB  0x05
4
5          .device ATmega328P
6          .org 0x00
7          rjmp reset
8
9          .org 0x34
10 reset:
11         ldi r16, bit(5)
12         out DDRB, r16
13         out PORTB, r16
14 loop:
15         rjmp loop
```

А теперь важная часть, которую необходимо понять, и без которой двигаться дальше будет весьма проблематично. Так выглядит Makefile для нашей простой программы:

```
1  AS = avra
2  PP = cpp
3
4  all: blink.hex
5
```

[1] стоит отметить, что avra не единственный ассемблер для контроллеров AVR

[2] и копилятор с языка ассемблера там тоже имеется, но по некоторым причинам мы не будем им пользоваться

```
6  blink.asm: blink.S
7          $(PP) -P blink.S blink.asm
8
9  blink.hex: blink.asm
10         $(AS) blink.asm $(CFLAGS) -o blink.hex
11
12 clean:
13         rm *.asm *.cof *.obj *.hex
14
15 .PHONY: clean
```

Сборка приложения состоит из двух частей: препроцессинг и компиляциz. В качестве препроцессора мы используем обычный C препроцессор со знакомыми вам директивами[1]. Полученный asm файл мы собираем используя avra.

В результате работы make вы получите ihex [Wikipedia, d] файл[2]. Именно файл в таком формате ожидает на вход программатор контроллера. В дальнейшем мы больше не будем возвращаться к Makefile-ам, подразумевая, что взяв описанный файл за образец, вы самостоятельно сможете написать новую версию со всеми необходимыми исправлениями.

Осталась самая малость, записать полученный файл blink.hex. Для этого нам понадобится вспомогательный скрипт detect.sh:

```
1  #!/bin/bash
2
3  # only udev systems supported so far
4  TOOL=udevadm
5
6  for DEV in /dev/ttyACM* ; do
7      if `$TOOL info --name=$DEV | grep -q 'arduino'` ; then
8          echo "$DEV"
9      fi
10 done
```

Для работы мы используем Arduino UNO (с подключенным к нему AVR Atmega 328p), в состав которого входит программатор, через который мы и загрузим программу на контроллер. Arduino UNO подключается к компьютеру через USB, и определеяется в системе как USB CDC ACM устройство (/dev/ttyACM*). Скрипт находит среди всех

[1] так привычнее, но для любителей avra предоставляет свои макросы

[2] и еще какие-то побочные файлы

имеющихся /dev/ttyACM тот, что соответствует Arduino UNO[3].

Последняя составная часть нашего пазла - скрипт загрузки программы на устройство upload.sh:

```
1 #!/bin/bash
2
3 BASE=`dirname $0`
4 DEVICE=`bash "$BASE/detect.sh" | head -n 1`
5 HEXFILE=$1
6 TYPE=ATMEGA328P
7
8 avrdude -F -V -c arduino -p $TYPE -P $DEVICE -b 115200 -U
    ↪ flash:w:$HEXFILE
```

Этот скрипт принимает в качестве параметра командной строки путь к hex файлу и вызывает avrdude, чтобы загрузить файл на устройство. Так же отметим, что скрипт upload.sh предполагает, что скрипт detect.sh находится в том же каталоге.

Если вам интересен смысл параметров avrdude, то вы можете почитать его документацию, мы не будем очень подробно останавливаться на параметрах. Опишем только самые основные:

- ключ -p - тип контроллера, в нашем случае все тот же Atmega 328p;
- ключ -P - файл устройства, который будет использовать avrdude для загрузки, это параметр определяется скриптом detect.sh;
- ключ -U - собственно описывает, что нужно сделать, в данном случае, flash - указывает на то, что мы работаем с flash памятью[1], w говорит о том, что мы хотим произвести операцию записи, а $HEXFILE - файл, который мы хотим записать.

Примерно выглядят вызов скрипта upload и его вывод при удачной загрузке программы:

```
1 kmu@kmu-tp-x230:~/ws/arduino/doc/avr/src$ sudo ./upload.sh
    ↪ blink/blink.hex
2 Swipe your right index finger across the fingerprint reader
3
4 avrdude: AVR device initialized and ready to accept
    ↪ instructions
```

[3]стоит заметить что вряд ли их будет много, ну разве что вы подключили несколько Arduino сразу

[1]есть еще fuse память, но нам это не особо важно

```
5
6  Reading |
       ↪ ############################################################ |
       ↪ 100% 0.00s
7
8  avrdude: Device signature = 0x1e950f
9  avrdude: NOTE: "flash" memory has been specified, an erase
       ↪ cycle will be performed
10           To disable this feature, specify the -D option.
11 avrdude: erasing chip
12 avrdude: reading input file "blink/blink.hex"
13 avrdude: input file blink/blink.hex auto detected as Intel
       ↪ Hex
14 avrdude: writing flash (40 bytes):
15
16 Writing |
       ↪ ############################################################ |
       ↪ 100% 0.02s
17
18 avrdude: 40 bytes of flash written
19
20 avrdude: safemode: Fuses OK (H:00, E:00, L:00)
21
22 avrdude done.  Thank you.
```

Если все прошло удачно, то можем считать, что все программы установлены и работают. После загрузки этой программы на Arduino UNO должен зажечься и постоянно гореть желтый светодиод рядом с выходом помеченным номером 13.

3 Структура программы

Мы научились собирать и загружать программу на контроллер, тем самым проверили работу toolchain-a. Теперь мы разберемся что из себя представляет программа, которую мы написали, почему она выглядит именно так, а не и иначе.

Начнем с простых вещей:

```
1          .device ATmega328P
```

Директива device сообщает ассемблеру конкретную модель AVR процессора [1]. Директива не транслируется в бинарный код, а просто

[1] а их не мало и между ними есть различия

сообщает ассемблеру дополнительную информацию [1].

```
1        .org 0x00
```

Еще одна директива[2] org, она указывает по какому смещению от начала сегмента кода[3] должны располагаться следующие за ней инструкции. По-умолчанию, первые инструкции и так начинаются по смещению 0x00, так что это не обязательная директива.

```
1        rjmp reset
```

Собственно, это первая инструкция нашей программы - инструкция относительного безусловного перехода. Переход в данном случае значит, что это инструкция передает управление какой-то другой инструкции[4].Безусловность, в данном случае, значит что этот переход не зависит от каких-либо условий и выполняется всегда, когда управдение передается этой инструкции. А относительность значит, что адрес инструкции, которой будет передано управление, определяется смещением относительно инструкции перехода. Это смещение часть кода команды, в случае инструкции rjmp, смещение - знаковое число, которое занимает 12 бит от кода команды[5]. Инструкции относительного безусловного перехода имеют в мнемоническом коде букву r в качестве префикса.

Небольшое лирическое отступление, в противовес инструкции rjmp, есть еще и инструкции условного перехода, которые, как правило, являются относительными. Есть в наборе команд AVR и инструкции безотносительного безусловного перехода, например, jmp, разница между rjmp и jmp заключается в том, что бинарный код команды jmp содержит адрес перехода, а не смещение и поэтому инструкция занимает больше места и может покрывать больший диапазон целевых адресов. Нередко, инструкцию rjmp можно заменить на jmp и наоборот без каких-либо последствий, но не в случае первой инструкии программы, мы еще вернемся к этому вопросу, когда будем говорить о прерываниях, сейчас же запомните, что первой инструкцией программы должна быть rjmp[6].

Итак, первая инструкция - инструкция перехода, но куда должен

[1] на самом деле, все будет работать и без этой директивы, так что это скорее для тех кто будет читать ваш код

[2] и опять не обязательная

[3] в данном случае от начала памяти, т. е. от адреса 0x00

[4] в противном случае, после завершения инструкции, управление передается следующей за ней инструкции

[5] используется дополнение до 2

[6] это не совсем правда, всех нетерпеливых и любопытных отсылаем к [AVR, a]

осуществляться этот переход? Ответ на этот вопрос лежит строкой ниже:

```
1         .org 0x34
2 reset:
```

Мы опять видим директиву org, теперь смещение равно 0x34. Дело в том, что первые несколько байт программы заняты адресами обработчиков прерываний, и первый адрес, не попадающий в эту таблицу обработчиков - 0x34, т. е. мы перепрыгиваем через таблицу векторов прерываний.

Собственно, начиная с метки reset и начинается по настоящему наша программа. Как вы уже могли догадаться метки - это способ задать имя для некоторого адреса в памяти, чтобы потом его можно было использовать совместно с какими-нибудь инструкциями перехода.

Основная логика программы начинается дальше:

```
1         ldi r16, bit(5)
2         out DDRB, r16
3         out PORTB, r16
```

Инструкция ldi (load immediate) загружает константу в регистр r16. Вообще в AVR есть довольно много регистров общего назначения (r0-r31), но не все они могут использоваться со всеми инструкциями, в частности для инструкции ldi тоже есть ограничения[1].

Immediate - это операнд, значение, которого сохраняется прямо в коде команды [2]. Так как immediate-ы занимают часть команды, то на них могут налагаться ограничения, например, операнд команды ldi должен быть от 0 до 255 включительно. В данном случае это не большое ограничение, так как все регистры общего назначения и так 8-битные.

В регистр r16 мы загружаем константу, в которой устновлен 5 бит, а остальные сброшены (т. е. 0x20 - мы считаем с 0). Не вдаваясь в подробности, мы хотим подать сигнал[3] на 5 ногу одного из портов ввода/вывода (в данном случае PORTB).

Но перед тем, как подать что-то на выход порта ввода/вывода, порт нужно сконфигурировать. Порты ввода/вывода AVR могут работать в разных режимах, в самом первом приближении, можете считать, что эти редимы - ввод и вывод [4]. По-умолчанию, все порты ввода/вывода работают в режиме ввода, нам нужен режим вывода, поэтому мы

[1] попробуйте поменять используемый регистр

[2] как например, адрес в инструкии jmp или смещение в инструкции rjmp

[3] логическую единицу

[4] очень логично

используем специальный регистр DDRB (регистр направления данных PORTB), который имеет адрес 0x04[1].

А чтобы подать логическую единицу на 5-ый бит PORTB мы опять записываем число 0x20, но уже в PORTB, который имеет адрес 0x05 в пространстве ввода/вывода. Строго говоря, мы не только записываем 1 в 5-ый бит, но и записываем 0 во все остальные - это важно понимать. Часто вам нужно будет сначала прочитать старое значение специального регистра или порта ввода/вывода, установить или сбросить в нем какой-то бит, и записать полученный результат назад.

Для записи значений по некоторым адресам в пространстве ввода/вывода мы используем специальную инструкцию out, есть и обратная инструкция in. Впринципе, можно использовать и обычные иснтрукции чтения и записи из памяти данных, но о том, как это делать мы поговорим позже.

```
1 loop:
2         rjmp loop
```

Финальный штрих. Вы уже должны понимать, что делают последние две строки программы. Они образуют бесконечный цикл. Зачем нужен бесконечный цикл? Если не завершить программу таким бесконечным циклом, то после выполнения последней инструкции out управление будет передано следующей за ней инструкции, которой нет. Т. е. процессор будет пытаться интерпретировать память за последней инструкцией как команды процессора, к чему это приведет не известно, поэтому мы в явном виде добавляем этот бесконечный цикл.

Осталось сделать одно финальное пояснение, в объяснении мы несколько раз говорили про адреса, например, адреса перехода или адрес специального регистра. Все это разные адреса в разных непересекающихся пространствах. Если мы говорим об адресе в контексте передачи управдения (вызова функции, ветвления в программе и тому подобное), то мы имеем ввиду адрес в памяти кода. Так же мы встретим еще и адрес в памяти данных, в отличие от знакомой вам x86, здесь адреса данных и адреса кода - это адреса в разных адресных пространствах, для работы с разными адресными пространствами используются разные инструкции. Это следствие Гарвардской архитектуры.

Объяснение первой самой простой программы довольно громозкое, но по мере освоения основ все станет проще. Далее мы подробнее

[1]это адрес в некотором специальном пространстве ввода/вывода, это не какая-то специфика AVR, отдельное адресное пространство ввода/вывода бывает и в других архитектурах

каснемся затронутой здесь темы портов ввода/вывод и пространства ввода/вывода, да и вообще организации памяти с точки зрения программы.

4 Организация памяти

Вы уже кое-что знаете о памяти в процессорах AVR. Сейчас мы подробно обсудим память контроллера с точки зрения программы. Как я уже упоминал, AVR является представителем гарвардской архитектуры, т. е. память команд и данных в нем разделены. С точки зрения программиста это разделение выглядит как разный набор инструкций для работы с памятью, в которой хранится код программы, и для работы с памятью, в которой хранятся данные. Кроме того доступ к программной памяти осуществляется словами по 2 байта, в то время как доступ к памяти данных побайтовый.

Для работы с программной памятью есть всего несколько инструкций LPM, ELPM - для чтения программной памяти, SPM - для записи программной памяти. Полное описание этих инструкций[1] вы можете найти в [AVR, b]. На этом мы остановим наш рассказ о программной памяти.

Гораздо больший интерес для нас представляет память для данных и пространство ввода/вывода. Начнем с более знакомой вам памяти для данных - это привычная нам память, в которой хранятся переменные программы, в этой же памяти организован стек и прочее.

Но что примечательно, в этой же памяти на самом деле хранятся и регистры процессора r0 - r31, они занимают адреса начиная с 0x00 по 0x1F включительно [2].

Следом за регистрами в память данных отображены порты ввода/вывода, т. е. пространство ввода/вывода отображено на память данных начиная с адреса 0x20. Что это значит для нас? Это значит, что вместо специальных инструкций in и out, работающих с пространством ввода/вывода, мы можем использовать обычные инструкции для работы с памятью, чтобы писать или читать порты ввода/вывода. Но мы должны преобразовать адрес в пространстве ввода/вывода в адрес в пространстве данных. Это преобразование выполняется, очевидно, прибавлением 0x20 к адресу в пространстве

[1] как и вообще всех инструкций AVR

[2] наверно, правильно говорить не, что они хранятся в этой памяти, а, что они отображены на эту память

ввода/вывода. Например, уже известную вам программу можно переписать не используя инструкцию out:

```
1  #define bit(x)       (1 << (x))
2  #define iospace(x)   (0x20 + (x))
3  #define DDRB         0x04
4  #define PORTB        0x05
5
6          .device ATmega328P
7          .org 0
8          rjmp    reset
9
10         .org 0x34
11 reset:
12         ldi r16, bit(5)
13         sts iospace(DDRB), r16
14         sts iospace(PORTB), r16
15
16 loop:
17         rjmp loop
```

Инструкция sts, как вы наверно уже догадались, в данном случае, записывает значение из регистра r16 по адресам 0x24 и 0x25, которые соответствуют DDRB и PORTB. Стоит отметить, что такое отображение пространства ввода/вывода в память не редкость, а скорее обычное дело для современных процессоров.

Пространство ввода/вывода отображено в память данных на адреса 0x20 - 0xff. Более того, обратится к некоторым адресам пространства ввода/вывода мы можем только через пространство данных, потому что инструкции in и out могут принимать только адреса от 0x00 до 0x3F[1], и этого диапазона явно не достаточно, чтобы покрыть все пространство ввода/вывод [2].

Вся остальная память начиная с адреса 0x0100 доступна для произвольных данных программы, например, в этой памяти организуют стек. Стек для программы - это просто указатель стека[3], который хранит просто некоторый адрес в памяти. Для работы со стеком есть пара инструкций push и pop. Инструкция push, записывает значение переданного регистра по адресу, который хранит указатель стека, а потом уменьшает значение указателя стека. Т. е. стек "растет вниз". Инструкция pop работает обратным образом, сначала она увеличивает

[1] всего 64 байта

[2] его размер посчитать не трудно, 0xff - 0x20 + 0x01 = 0xe0, т. е. 224 байта

[3] который занимает адреса 0x3d и 0x3e в пространстве ввода/вывода

значение указателя стека, потом достает значение по указанному адресу и сохраняет его в регистре.

Перед тем как работать со стеком его нужно инициализировать. Обычно указатель стека инициализируют так, чтобы он указывал на последний адрес памяти[1]. Для процессоров Atmega 328p максимальный адрес 0x08ff, т. е. 0x0100 байт на которые отображены регистры и пространство ввода/вывода плюс еще 0x0800 байт. Не трудно заметить, что это число не поместится в 8-битный регистр, т. е. указатель стека состоит из двух байт.

Стек используется некоторыми инструкциями процессора неявно, например, при вызове функций[2], или в обработчиках прерываний[3].

Давайте рассмотрим пример работы со стеком, а кроме того напишем нашу первую настоящую функцию. Но предварительно мы вынесем все наши макроопределения в файл atmega328.h:

```
1  #define RAMEND      0x08FF
2  #define DATABEGIN   0x0100
3  #define DDRB        0x04
4  #define PORTB       0x05
5  #define SPH         0x3e
6  #define SPL         0x3d
7
8  #define high(x)     (((x) & 0xff00) >> 8)
9  #define low(x)      ((x) & 0x00ff)
10 #define iospace(x)  ((x) + 0x20)
```

Хардкодить везде адреса не самая светлая идея, поэтому мы ввели с помощью препроцессора несколько мнемонических обозначений. Например, мы ввели обозначение RAMEND для последнего доступного адреса в памяти, завели обозначения DDRB и PORTB для одноименных регистра и порта, SPH и SPL - обозначения для адресов старшего и младешго байта указателя стека в пространстве ввода/вывода.

Кроме того мы определили несколько макро-функций, например, high и low достают старший и младший байт 2-батовой константы, соответственно. Мы используем их, чтобы разделить RAMEND на верхний и нижний байты. Кроме того мы завели макро-функцию iospace для преобразования адреса в пространстве ввода/вывода в адрес в пространстве данных, мы не будем пользоваться ей очень часто, для нас она скорее является напоминанием. Ну и еще одна полезная макро-

[1] никто не мешает вам сделать по другому, иногда это даже полезно

[2] имеется ввиду специальная инструкция, а не уже известная вам rjmp

[3] кроме RESET, который является особым случаем

функция bit возвращает значение, в которой установлен бит с заданным номером. Будьте внимательны это макро-функции, они обрабатываются препроцессором и их не существует в момент исполнения программы.

Теперь используя введенные обозначения мы инициализируем стек, а зажигание светодиода вынесем в отдельную функцию:

```
1  #include "atmega328.h"
2
3          .device ATmega328P
4          .org 0
5          rjmp reset
6
7          .org 0x34
8  reset:
9          ; setup stack
10         ldi r16, high(RAMEND)
11         out SPH, r16
12         ldi r16, low(RAMEND)
13         out SPL, r16
14
15         rcall light_portb5
16
17 loop:
18         rjmp loop
19
20 ; it is our first real function
21 light_portb5:
22         ldi r16, bit(5)
23         out DDRB, r16
24         out PORTB, r16
25         ret
```

Теперь остановимся подробнее на функции light_portb5. Нет нас не интересует, что делает эта функция - вы уже должны сами понимать. Нас интересует, что именно делает этот участок кода полноценной функцией. Нас интересуют две новые инструкции rcall и ret. Инструкция rcall - относительный вызов функции. Относительный в данном случае значит то же самое, что и для инструкции rjmp, т. е. адрес перехода определяется как адрес инструкции rcall плюс некоторое смещение, которое хранится прямо в коде команды. Есть и обычная инструкция call, в которой целевой адрес закодирован непосредственно. Практическая разница между двумя этими инструкциями в том, что бинарный код команды rcall занимает 2 байта, а бинарный код команды

call 4 байта. Соответственно, диапазон адресов доступных команде rcall немного меньше чем диапазон адресов доступных команде call[1].

Так зачем же нужная инструкция rcall, если уже есть rjmp? Разница между этими двумя инструкциями в том, что rcall не только передает управление по указанному адресу, но и сохраняет на стеке адрес возврата, т. е. адрес инструкции следующей за инструкцией rcall[2]. Зачем это нужно? Очевидно, для того, чтобы функция знала куда передавать управление, когда она закончит работу. Собственно, ровно этим и занимается инструкция ret в конце функции, она берет со стека два байта интерпретирует их как адрес в программной памяти и передает управление по этому адресу. Таким образом одна и та же функция может вызываться из разных мест. Стоит отметить, что перед инструкцией ret нужно вернуть указатель стека в то положение, в котором он находился в самом начале работы функции, или проще говоря, нужно убедиться, что два байта на вершине стека - это именно те два байта, которые туда положила инструкция rcall, а не данные, которые на стек положила наша функция[3].

5 Инструкции процессора AVR

До сих пор мы видели всего несколько простых инструкций процессора AVR. Теперь мы уделим чуть больше внимания, собственно, набору команд AVR.

5.1 Арифметические действия

Начнем с самого очевидного - сложения. Для сложения в процессорах AVR есть две базовые инструкции[4]:

- add (add without carry) - принимает два регистра как аргументы и сохраняет сумму значений регистров в первом из них;
- adc (add with carry) - принимает два регистра как аргументы и сохраняет сумму регистров и значения флага C регистра SREG в первом из них.

[1] на некоторых процессорах AVR rcall может адресовать всю программную память, а на некоторых нет

[2] или call

[3] в данном конкретном примере мы не используем стек внутри функции, так что и проблемы такой нет

[4] на самом деле их чуть больше, но мы не ставим себе задачу покрыть все множество инструкций AVR, для этого есть [AVR, b]

Figure 5.1: Регистр SREG

Перед тем как разбираться подробнее с инструкциями add и adc, давайте поговорим о флаговом регистре SREG[1]. SREG (Status Register) содержит некоторое количество флагов fig. 5.1. Эти флаги выставляются или сбрасываются в результате работы инструкций процессора и используются для работы другими инструкциями процессора.

Среди флагов регистра SREG есть, например, следующие:

- C (Carry Flag) - флаг переноса выставляется, если в результате арифметического действия произошло беззнаковое переполнение;
- Z (Zero Flag) - флаг выставляется, если результатом действия был 0;
- V (Overflow Flag) - флаг знакового переполнения;
- N (Negative Flag) - если интерпретировать результат действия как знаковое число, то этот флаг выставляется, если это число было отрицательным, и сбрасывается в противном случае.

Так вот, возвращаясь к инструкциям сложения. Инструкция add выставляет флаги Z, C, N, V[2], в зависимости от результата сложения. Например, если результат не помещается в 8 бит, то в результате выполнения инструкции будет выставлен флаг C.

Инструкция adc выставляет те же самые флаги, что и add, но при этом еще учитывает значение флага C от предыдущих операций. Используя пару инструкций add и adc мы можем выполнять арифметические действия над числами большей битности. Например, следующая функция складывает 2 16-битных числа переданных в регистрах r18-r19 и r20-r21, а результат сложения сохраняется в регистрах r18-r19[3]:

```
1 __add16:
2         add r18, r20
3         adc r19, r21
4         ret
```

[1]флаговые регистры есть, наверно, во всех процессора, так что уделите этому некоторое время

[2]и не только

[3]калька с поведения инструкции add

Аналогичным образом мы можем работать над 32-битными числами:

```
1  __add32:
2          add r18, r22
3          adc r19, r23
4          adc r20, r24
5          adc r21, r25
6          ret
```

Обратите внимание, что эти инструкции работают как с беззнаковыми числами, так и со знаковыми, так как бинарное представление результат сложения не зависит от того, работаем мы со знаковыми числами или беззнаковыми, это приятная особенность дополнения до 2.

Для вычитания тоже есть пара аналогичных инструкций:

- sub (Sub Without Carry) - принимает два регистра как аргументы и сохраняет разность значений регистров в первом из них;
- sbc (Sub With Carry) - аналогично sub, но вычитает из результата значение флага C.

При вычитании флаг C выставляется, если произошел заем[1]. Это легко объясняется, если свести вычитание к сложению. Если при вычитании должен произойти заем, то при сложении произойдет переполнение, и наоборот. Собирая все вместе, так выглядят функции вычитания 16-битных и 32-битных чисел:

```
1  __sub16:
2          sub r18, r20
3          sbc r19, r21
4          ret
5
6  __sub32:
7          sub r18, r22
8          sbc r19, r23
9          sbc r20, r24
10         sbc r21, r25
11         ret
```

Есть еще пара полезных инструкций, которые имеют отношение к базовой арифметике:

- inc - увеличивает значение регистра на 1;

[1] имеется ввиду из "виртуального" 8 бита, помните, что мы считаем с 0

- dec - уменьшает значение регистра на 1.

Инструкции inc и dec так же как и add/adc/sub/sbc выставляют или сбрасывают почти те же флаги в регистре SREG, в зависимости от результата. Для каждого флага в SREG есть формула, но зачастую все флаги можно интерпретировать по смыслу[1].

В процессорах AVR есть инструкции для умножения и работы с дробными числами, но мы не будем их касаться потому что:

1. они нам не понадобятся;
2. вы всегда сможете прочитать о них в [AVR, b].

5.2 Битовые операции

В [AVR, b] инструкции о которых пойдет речь называют логическими, но нам привычнее называть их битовыми[2].

Собственно вот базовый набор доступных битовых операций:

- and - принимает два регистра как операнды, сохраняет результат побитового И в первом из них;
- andi - то же самое, что и and, но второй операнд не регистр, а непосредственное число (immediate);
- or - побитовое ИЛИ двух регистров;
- ori - побитовое ИЛИ регистра и immediate-а;
- eor - исключающее ИЛИ двух регистров;

С этими операциями все просто и понятно, они как и арифметические операции выставляют биты SREG кроме C. Есть в AVR еще несколько интересных битовых инструкций:

- sbr (Set Bit(s) in Register) - принимает регистр и маску k, делает побитовое ИЛИ регистра и маски;
- cbr (Clear Bit(s) in Register) - принимает регистр и маску k, сбрасывает в регистре биты установленные в k.

Внимательный читатель мог отметить, что инструкция sbr делает то же самое, что и инструкция ori - и это правда. А вот инструкция cbr чуть сложнее, и она может может оказаться полезной.

[1] как было с флагом C

[2] для людей знакомых с C наш вариант должен быть понятнее

5.3 Инструкции перехода

Вы уже знакомы с базовой арифметикой, теперь вам нужно узнать как организовать поток управления[1]. Вы уже знакомы с некоторыми конструкциями управления: rjmp, rcall, ret - все они приводят к передаче управления другому участку программы. Но сейчас мы сосредоточимся именно на передаче управления с проверкой условия. Для этого мы сначала узнаем как можно работать с условиями на языке ассемблера AVR.

Собственно, на самом деле вы уже знаете как выглядят условия в процессоре AVR [2]. Условия - это значения флагов регистра SREG. В зависимости от значений этих флагов мы можем осуществлять ветвления в программе. Но перед тем как мы перейдем к инструкциям условного перехода, мы познакомимся еще с несколькими полезными инструкциями, которые устанавливают флаги SREG:

- tst - инструкция принимает регистр как аргумент и делает побитовое И регистра с самим собой, выставляет флаги SREG в зависимости от полученного результата, затем отбрасывает полученный результат, т. е. как инструкция and, но не сохраняет полученный результат в регистре;
- cp (Compare) - принимает два регистра, как аргументы, вычитает значение второго регистра из первого, выставляет флаги в зависимости от полученного результата, после чего, как и tst, отбрасывает полученный результат;
- cpc (Compare With Carry) - как cp, только еще учитывает значение флага C;
- cpi (Compare With Immediate) - как cp, только сравнивает значение регистра с константой.

Эти инструкции полезны при проверке условий, например, с использованием cp, можно сравнивать числа на больше/меньше. Как использовать установленные флаги SREG, чтобы осуществлять ветвление в программе? Для этого существует целый набор инструкций, каждая из которых проверяет один из флагов в регистре SREG:

- breq (Branch if Equal) - принимает как аргумент метку для перехода, и если установлен флаг Z, то передает управление этой

[1] как реализовать знакомые вам if/for/while

[2] и не только в AVR

метке, иначе передает управление следующей инструкции; логика названия довольно простая, если вы использовали инструкцию cp для сравнения двух чисел, то если два числа равны, то результатом их вычитания будет 0, значит в результате инструкции cp будет установлен флаг Z регистра SREG, в противном случае он будет сброшен;

- brne (Branch if Not Equal) - инструкция "обратная" breq, т. е. переход будет осуществлен, если флаг Z сброшен;
- brsh (Branch if Same or Higher) - осуществляет переход, если флаг C сброшен, т. е. если мы исплользуем cp и левый операнд больше или равен правому, то в результате беззнаково вычитания не произойдет заема, а значит флаг C будет сброшен и произойдет переход;
- brlo (Branch if Lower) - инструкция "обратная" brsh;
- brge (Branch if Greater or Equal) - как brsh, только для знаковых чисел;
- brlt (Branch if Less Than) - инструкция обратная brge.

На самом деле инструкций перехода несколько больше, за деталями, как обычно, отправляйтесь читать [AVR, b].

Обратите внимание, что инструкции условного перехода разделяются на знаковые и беззанковые, дело в том, что переполнение знакового числа и переполнение беззнаковго числа - это не одно и то же, и соответствуют они разным флагам в SREG, даже не смотря на то, что битовое представление результата не зависит от знаковости.

Кроме Branch инструкций в AVR есть еще Skip инструкции. Они проверяют те же самые условия, но если условие выполнено, они пропускают следующую инструкцию, а не передают управление по метке.

В качестве примера, давайте посмотрим на функцию вычисления чисел Фибоначчи:

```
1 fib:
2         ; uint16_t cur = 0;
3         ldi r18, 0x00
4         ldi r19, 0x00
5
6         ; uint16_t nxt = 1;
```

```
 7           ldi r20, 0x01
 8           ldi r21, 0x00
 9
10           ; while (n--) {
11 fib_cond:
12           subi r22, 1
13           brcs fib_out
14
15           ; uint16_t tmp = nxt;
16           mov r24, r20
17           mov r25, r21
18
19           ; nxt += cur
20           add r20, r18
21           adc r21, r19
22
23           ; cur = tmp
24           mov r18, r24
25           mov r19, r25
26
27           ; }
28           rjmp fib_cond
29
30           ; return cur
```

Здесь нужно сделать несколько пояснений, во-первых, аргументом функции является 8-битное число n, которое передается в регистре r22. Результатом функции является 16-битное число, которое возвращается в регистрах r18-r19. Все остальные регистры используются для временных значений.

Стоит отметить, что эта функция портит значение своего параметра, а кроме того использует регистры r20-r22 и r24-r25 не восстанавливая их исходные значения - это не всегда приемлемое поведение. Обычно правила передачи и возврата значений определяются так называемой конвенцией вызова. В эти правила так же входит описание назначения и способа использования различных регистров функциями. В частности какие регистры функция может использовать не восстанавливая оригинальные значения при завершении (Clobbered Registers или Non-volatile), а какие регистры она должна восстановить в исходное значение перед завершением (Nonclobbered Registers или Volatile). Разные компиляторы и разные архитектуры используют разные конвенции вызова, например, в 32-битном x86, параметры передаются в функцию,

как правило, через стек, в то время как в 64-битном x86, их чаще передают через регистры.

Возвращаясь назад к функции, как говорят комментарии, регистры r18-r19 и r20-r21 хранят временные значения cur и nxt, соответственно. cur хранит текущее число Фибоначчи, а nxt - следующее.

Обратите внимание на то, как проверяется условие цикла - мы сначала декрементируем значение r22 (которое хранит параметр функции), а потом проверяем флаг C в инструкции brcs. Как это работает? Если r22 равен 0, то уменьшение на единицу приведет к переполнению, т. е. будет установлен флаг C. В противном случае флаг C будет сброшен.

Далее идет незнакомая вам инструкция mov. Она копирует содержимое его второго операнда в первый. Используя две таких инструкции[1] мы копируем значение nxt[2] во временное хранилище - регистры r24-r25.

Далее мы складываем два значения, используя уже известную вам последовательность add + adc.

Наконец мы переносим сохраненное значение nxt в cur, тем самым завершая итерацию цикла. Следующая инструкция rjmp просто возвращает нас к проверке условия цикла. Таким образом, когда значение r22 перейдет через 0, и станет равным 255[3], в регистрах r18-r19 окажется нужное число Фибоначчи.

5.4 Инструкции передачи данных

На инструкциях передачи данных мы не будем подробно останавливаться, потому что большинство из них не требует никаких пояснений. Более того вы уже видели некоторые из них, например, sts, ldi, in, out или mov. Полный список как всегда доступен в [AVR, b] - не поленитесь по крайней мере просмотреть список доступных инструкций[1].

5.5 Используем objdump

Иногда не сразу удается придумать простой способ проверки сложного условия цикла, или что-то в этом роде. Да и вообще не всегда

[1] можно обойтись одной инструкцией movw

[2] которое хранится в r20-r21

[3] помните, мы проверяем переполнение

[1] он даже специально сведен в удобную для использования табличку, дальше этой таблички заглядывать приходится очень не часто

хочется зарываться в язык Ассемблера[2]. В этом случае мы можем воспользоваться помощью компилятора.

Для этого просто напишите программу на языке C:

```
1  #include <stdint.h>
2
3  int16_t sadd(int16_t a, int16_t b)
4  {
5          return a + b;
6  }
7
8  int16_t ssub(int16_t a, int16_t b)
9  {
10         return a - b;
11 }
12
13 uint16_t uadd(uint16_t a, uint16_t b)
14 {
15         return a + b;
16 }
17
18 uint16_t usub(uint16_t a, uint16_t b)
19 {
20         return a - b;
21 }
```

Скомпилируйте этот файл используя avr-gcc, и вы получите объектный файл, пусть он называется arith.o. Тогда используя утилиту avr-objdump, мы можем посмотреть ассемблерный код:

```
1  avr-objdump -D arith.o
2
3  arith.o:     file format elf32-avr
4
5
6  Disassembly of section .text:
7
8  00000000 <sadd>:
9     0:   86 0f           add     r24, r22
10    2:   97 1f           adc     r25, r23
11    4:   08 95           ret
12
13 00000006 <ssub>:
```

[2]это может быть довольно утомительным занятием

```
14     6:   86 1b           sub     r24, r22
15     8:   97 0b           sbc     r25, r23
16     a:   08 95           ret
17
18 0000000c <uadd>:
19     c:   86 0f           add     r24, r22
20     e:   97 1f           adc     r25, r23
21    10:   08 95           ret
22
23 00000012 <usub>:
24    12:   86 1b           sub     r24, r22
25    14:   97 0b           sbc     r25, r23
26    16:   08 95           ret
27
28 Disassembly of section .comment:
29
30 00000000 <.comment>:
31     0:   00 47           sbci    r16, 0x70       ; 112
32     2:   43 43           sbci    r20, 0x33       ; 51
33     4:   3a 20           and     r3, r10
34     6:   28 47           sbci    r18, 0x78       ; 120
35     8:   4e 55           subi    r20, 0x5E       ; 94
36     a:   29 20           and     r2, r9
37     c:   34 2e           mov     r3, r20
38     e:   38 2e           mov     r3, r24
39    10:   32 00           .word   0x0032  ; ????
```

В этом огромном выводе можно увидеть метки с именами, которые соответствуют нашим функциям. Но чтобы получить такой компактный вывод требуется компилировать код с оптимизациями, например, данный вывод получен при компиляции с флагом -O3.

Стоит отметить, что правильные метки перехода расставляются в момент компоновки, а не в момент компиляции, поэтому инструкции ветвления обычно содержат какую-то невнятную информацию о том, куда должен осуществляться переход.

Еще один важный момент в том, что в результате сильных оптимизаций можно получить ассемблерный код, который будет довольно трудно сопоставить с исходным кодом на языке C. Есть два способа избежать такой проблемы:

- отключить оптимизации компилятора - в этом случае перевод C кода в ассемблерный код будет довольно прямолинейным, но вместе с тем в коде будет присутствовать много лишних действий, и

выделить среди них нужную часть может быть немного трудно;

- сохраняйте функции маленькими - чем меньше размер С функции, тем меньше у компилятора возможностей для оптимизации, это решение пожалуй самое лучшее.

Альтернативой утилите objdump может стать флаг -S компилятора gcc. Результатом компиляции с флагом -S будет не объектный или исполняемый файл, а файл с ассемблерным кодом. В этом коде будет большое количество атрибутов, о которых мы не говорили, но понять их смысл не трудно, впрочем как и не трудно их проигнорировать.

6 Самостоятельная работа: моргаем светодиодами

Вы уже знаете как зажигать и тушить светодиоды, теперь сделаем огромный шаг вперед - научимся моргать светодиодами. Чтобы моргание было заметно человеческому глазу необходимо зажигать светодиод на некоторое продолжительное время, затем тушить его на некоторое время, и повторять это в цикле.

Порядок выполнения задания:

1. Реализуйте функцию loop, которая, как видно из названия, представляет из себя цикл. Тело цикла должно выполнится ровно 255 раз. Не важно что именно будет делать этот цикл - задача этой функции просто выполнятся некоторое время.

2. Посчитайте время, которое выполняется функции loop. Для подсчета времени используйте информацию из [AVR, b] и значение частоты тактового генератора (16МГц для Arduino). Чем точнее вы посчитаете тем лучше (считайте, что функцию вызывается с помощью rjmp).

3. Реализуйте еще одну функцию с именем delay. Эта функция принимает как параметр 16 битное число. Младшие 8 бит в регистре r24, а старшие 8 бит в регистре r25. Функция delay должна вызывать функцию loop ровно столько раз, сколько указано в регистра r24-r25.

4. Посчитайте, при каких значениях r24-r25 функция delay будет выполнятся примерно 1 с. Постарайтесь получить максимальную точность (желательно, не на глаз).

5. Напишите программу, которая используя функцию delay моргает встроенным светодиодом с периодом в 2 с. Т. е. зажигает светодиод на 1 с, потом гасит светодиод на 1 с, и так в бесконечном цикле.

Требования к решению минимальны:

1. не нужно подбирать задержки на глаз, т. е. готовьтесь рассказать, как вы рассчитали нужные значения параметров для delay;

2. код программы будут смотреть[1], так что постарйтесь соблюдать какое-то осмысленное форматирование;

Для любителей поиграться со свободным временем. В задании используется встроенный светодиод, но вы можете попробовать подключить внешний светодиод[2] к тому же 13 выходу, или к какому-то другому[3].

При подключении внешнего светодиода учтите, что свтодиоды штука хлипкая и капризная, поэтому запомните несколько правил "пальца":

- подключайте светодиоды через сопростивление (конкретное сопротивление зависит от светодиода, но для наших хватит 220 Ом);

- не включайте светодиоды параллельно (одинаковые на первый взгляд светодиоды, на самом деле обладают разными характеристиками), т. е. используйте для каждого светодиода отдельную ногу контроллера и свой резистор.

7 Таймеры и прервыания

Вы написали hello world программу для контроллера[1]. Попутно вы познакомились с портами ввода/вывода, пространством ввода/вывода, отображением пространства ввода/вывода на основную память - все это

[1] хотя и довольно бегло

[2] он побольше, да и цвет не такой противный

[3] или нескольким сразу

[1] имеется ввиду мигание светодиодом

важные концепции, применимые не только в AVR контроллерам, но и к более мощным процессорам. Теперь пришло время познакомится с еще одной важной вещью - прерываниями. Знакомится с ними мы будем на примере таймеров/счетчиков процессора AVR.

Процессоры atmega328 имеют три таймера/счетчика[2]. Таймеры/счетчики могут работать в разных режимах, нас будет интересовать только один из них - CTC. Что из себя представляет режим CTC я объясню чуть дальше.

С каждым таймером/счетчиком ассоциирован регистр - счетчик[3]. Этот регистр инкрементируется[4] на каждый тик[5].

Как часто тики происходят? Этот параметр настраивается. Наибольшая скорость определяется системным тактовым генератором[6]. В Arduino используется внешний генератор тактов с частотой 16МГц[7].

Таймеры 0 и 2 являются 8-битными, т. е. регистр счетчик состоит из 8 бит, а таймер 1 - 16-битный. Если тики генерируются с частотой 16МГц, то переполнение 8-битного счетчика будет происходить каждые 16мкс, а переполнение 16-битного счетчика будет происходить каждые 4.096мс - это нужно учитывать, если вы собираетесь как-то использовать значение счетчика.

AVR процессоры предоставляют возможность управлять частотой генерации тиков с использованием схемы - делителя частоты[8]. В AVR частоту генерации тиков можно уменьшить в 1, 8, 64, 256 и 1024. Т. е. если частота системного генератора 16МГц, то можно генерировать тики на частотах 16МГц, 2МГц, 250кГц, 62.5кГц и 15.625кГц. По факту, делитель частоты просто пропускает каждый 1/8/64/256/1024 тик, замалчивая все остальные. Управление коэффициентом деления осуществляется с помощью бит специального регистра, о котором мы поговорим позже в примерах.

Даже несмотря на наличие делителя диапазон рабочих частот таймера весьма ограничен[1]. Для преодоления этой небольшой трудности существует набор специальных прерываний таймера, которые срабатывают, когда значение счетчика будет равно какому-то наперед

[2]будем нумеровать их начиная с 0, как обычно

[3]поэтому мы и называем их таймерами/счетчиками

[4]или декрементируется, так тоже можно

[5]tick - на русском, наверно, правильно говорить сигнал таймера, но тик короче

[6]он может быть внешним или внутренним, т. е. разной частоты, поэтому полезно знать каким именно тактовым генератором вы пользуетесь

[7]период такого генератора 63нс - что довольно не много

[8]на английском языке ее называют prescaler

[1]хотя этого уже достаточно, чтобы организовать довольно точный подсчет времени

заданному числу. После срабатывания такого прерывания значение счетчика будет сброшено в 0 на следующем тике, и подсчет пойдет по новой - это и есть режим CTC. Как это можно использовать?

Допустим, мы хотим, чтобы некоторый обработчик вызывался с частотой 1Гц. У нас в распоряжении 16-битный таймер/счетчик. Так как частота 1Гц заметно меньше 16МГц системного таймера, то мы сразу задаем самый большой коэффициент деления из возможных - 1024, т. е. частота генерации тиков будет равна 15.625кГц. Это значит, что значение таймера счетчика увеличивается 15625 раз в секунду. Т. е. через 15625 тиков пройдет примерно 1с времени. Т. е. если мы установим значение для сравнения равным 15624[2], то прерывание будет генерироваться каждую секунду. Заметьте, что мы должны использовать именно 16-битный таймер/счетчик, в противном случае у нас просто не хватит бит, чтобы записать в регистр для сравнения число 15624. В общем случае, значение для сравнения можно определить следующим образом:

$$V_{comp} = \frac{F_{sys}}{K_{scale} \times F_{req}} - 1 \tag{1}$$

- V_{comp} - значение для сравнения;
- F_{sys} - частота системного генератора тактов, 16МГц в нашем случае;
- K_{scale} - коэффициент делителя частоты, в примере выше используется 1024;
- F_{req} - желаемая частота генерации прерывания, в примере выше используется 1Гц.

С использованием этих прерываний таймера мы можем довольно точно измерять небольшие интервалы времени. Единственное ограничение - это количество таймеров/счетчиков - их всего три[3]. В наших примерах хватит и двух таймеров счетчиков[4], так что заморачиваться по поводу этого нам не придется. Но и поверх этих трех таймеров можно организовать довольно сложный универсальный учет времени, и пример, который вы увидите дальше, дает некоторую наводку, как это можно сделать[1].

[2]плюс 1 тик на сбрасывание значения в 0

[3]и тут не все гладко, два из них не совсем независимы

[4] один 8-битный и один 16-битный

[1]он специально усложнен чтобы претендовать на хоть на какую-то общность

Итак перейдем к примеру. Для этого мы напишем[2] программу, которая моргает светодиодом. Мы опять же будем использовать функцию, которая выполняется некоторое продолжительное время, но, на этот раз, функция задержки будет использовать таймер/счетчик.

Для начала, мы заведем место для глобального счетчика тактов[3]. Это 4-байтовое число, которое будет хранится по адресу 0x0100 [4]:

```
1 #define DATABEGIN   0x0100
```

```
1 #define JIFFIES     DATABEGIN
```

Мы хотим настроить таймер так, чтобы прерывание срабатывало каждую мс, т. е. нам нужны прерывания с частотой 1000Гц[5]. Примем в качестве коэффициента деления 64[6]. По формуле eq. 1, число для сравнения должно быть 249. Число как раз помещается в 8-битный регистр, поэтому мы будем использовать 8-битный таймер/счетчик 0.

Работой таймера/счетчика 0 управляют регистры TCNT0, TCCR0A, TCCR0B, OCR0A, OCR0B и TIMSK0[7]. Заведем для этих регистров именованные макроопределения:

```
1 #define TCCR0A      0x24
2 #define TCCR0B      0x25
3 #define TCNT0       0x26
4 #define OCR0A       0x27
5 #define OCR0B       0x28
6 #define TIMSK0      0x4e
```

Адреса, как и положено, были взяты из [AVR, a]. Итак, зачем нужна вся эта куча регистров? Начнем с TCNT0 (Timer Counter) - это собственно и есть счетчик, который инкрементируется на каждый тик таймера. Мы можем писать и читать значение этого счетчика.

OCR0A и OCR0B - 8-битные регистры, которые хранят значения для сравнения. В них хранятся те самые значение, с которыми сравнивается значение счетчика. В режиме CTC нас интересует только OCR0A. Как

[2] в очередной раз

[3] аля Linux jiffies

[4] первый незанятый адрес памяти данных

[5] 1 мс не очень большое значение, что позволит нам измерять время достаточно точно, с другой стороны по сравнению с частотой тактового генератора это значение просто огромно, что позволит не думать о времени выполнения обработчика прерываний

[6] попробуйте другие коэффициенты и вы увидите почему я выбрал именно такой

[7] да их не мало, но все не так страшно

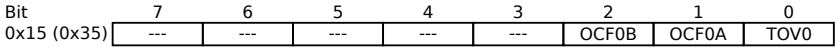

Figure 7.2: Регистр TIFR0

только значение TCNT0 сравняется с OCR0A, в регистре TIFR0 (Timer/-Counter 0 Interrupt Flag Register fig. 7.2) выставляется флаг OCF0A (Timer/Counter 0 Output Compare A Match Flag). Само по себе это событие не приводит к возникновению прерывания.

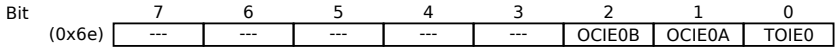

Figure 7.3: Регистр TIMSK0

Чтобы при установке флага OCF0A было сгенерировано прерывание необходимо установить флаг OCIE0A (Timer/Counter 0 Output Compare Match A Interrupt Enable) в регистре TIMSK0 (Timer/Counter 0 Interrupt Mask Register fig. 7.3). Вообще регистр TIMSK0 отвечает за включение/выключения различных прерываний связанных с таймером 0. Аналогичные регистры есть и для других таймеров.

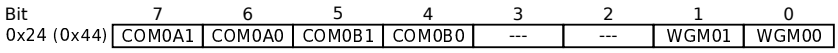

Figure 7.4: Регистр TCCR0A

Ну и наконец регистры TCCR0A (Timer/Counter 0 Control Register A fig. 7.4) и TCCR0B (Timer/Counter 0 Control Register B fig. 7.5) отвечают за режим работы таймера и коэффициент деления частоты. Чтобы перевести таймер в режим CTC[1] нужно сбросить бит WGM00 и установить бит WGM01 в регистре TCCR0A, и сбросить бит WGM03 в регистре TCCR0B [2].

Чтобы установить коэффициент деления равным 64, нужно установить биты CS00 и CS01 в регистре TCCR0B, и сбросить в этом же регистре бит CS02. Кстати, если сбросить все эти биты (CS00-CS02), то таймер будет остановлен.

Собираем все в кучу. Алгоритм действий для инициализации нашего счетчика времени следующий:

[1] а это единственный режим таймера, который нас интересует

[2] смотрите на группу WGM0x как на единую группу бит, а на регистры TCCR0A и TCCR0B как на две части одного 16-битного регистра, собственно, они даже в памяти располагаются последовательно

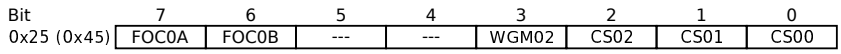

Figure 7.5: Регистр TCCR0B

1. сбрасываем значение глобального счетчика JIFFIES в 0;
2. сбрасываем значение TCNT0 в 0;
3. записываем в регистр TCCR0A значение 0x02 (тем самым устанавливаем бит WGM01 и сбрасываем все остальные);
4. записываем в регистр TCCR0B значение 0x03 (тем самым устанавливаем биты CS00 и CS01, а остальные сбрасываем в 0);
5. записываем в регистр OCR0A значение 0xf9 (249);
6. записываем в регистр TIMSK0 значение 0x02 (тем самым устанавливаем бит OCIE0A, а остальные сбрасываем в 0);

На деле все оказывается не так уж и сложно, давайте посмотрим на функцию, которая делает все описанное выше:

```
1  setup_jiffies:
2          push r16
3
4          clr r16
5          sts JIFFIES, r16
6          sts JIFFIES + 1, r16
7          sts JIFFIES + 2, r16
8          sts JIFFIES + 3, r16
9
10         sts iospace(TCNT0), r16
11         ldi r16, bit(WGM01)
12         sts iospace(TCCR0A), r16
13         ldi r16, (bit(CS00) | bit(CS01))
14         sts iospace(TCCR0B), r16
15         ldi r16, 0xf9
16         sts iospace(OCR0A), r16
17         ldi r16, bit(OCIE0A)
18         sts iospace(TIMSK0), r16
19
20         pop r16
21         ret
```

Обратите внимание, что мы используем макрос iospace и команду sts, для записи в регистры. Дело в том, что записать что-либо в регистр TIMSK0 используя команду out не получится, так как он находится за пределами допустимого диапазона адресов команды out. Поэтому мы вынуждены использовать sts для записи в него, и используем sts для записи в другие регистры для однообразия[1].

Мы умеем настраивать таймер так, чтобы с частотой в 1кГц срабатывало прерывание известное в [AVR, a] под именем TIMER0 COMPA. Теперь нам нужен обработчик, который будет вызываться при этом прерывании. Вы уже знаете, что адреса всех обработчиков прерываний располагаются в самом начале программы. Нужному нам прерыванию соответствует адрес 0x1c, т. е. туда мы должны поместить инструкцию rjmp, которая передаст управление коду, который обработает прерывание.

Все что будет делать обработчик прерывания это загружать в регистры 4 байтовое значение из адреса 0x0100 (JIFFIES). Далее он увеличивает это значение на 1 и сохраняет полученный результат назад. Концептуально все просто, но обработчик прерывания - функция не совсем обычная. Она вызывается независимо от основного потока управления, т. е. прерывание может произойти в любой момент времени, вклиниться между любыми последовательными инструкциями.

Давайте подумаем, к чему это может привести. Рассмотрим, например, уже известный вам код сложения двух 16-битных чисел:

```
1 __add16:
2         add r18, r20
3         adc r19, r21
4         ret
```

Представим, что после выполнения[2] инструкции add, но до того как выполнилась инструкция adc произошло прерывание. Выполнение основного кода программы было прервано, и управление передали обработчику прерывания. Обработчик прерывания в свою очередь записал в регистры нужные ему значения. При этом он мог испортить значения регистров r19 и r20, которые были в функции __add16. После завершения обработчика прерывания управление вернулось назад к инструкции adc функции __add16, но значения регистров r19 и r20 стали совсем другими. Я уже упоминал о подобной проблеме в контексте

[1] вам повторять такое не обязательно

[2] это не совсем корректно, представлять все именно так, но от полной корректности суть рассуждения не изменится, только детали станут сложнее

функций. Принципиальное отличие от ситуации с функциями в том, что в случае с функциями у нас было два варианта решения:

- перед вызовом функции сохранить все нужные нам значения;
- написать функцию так, чтобы она не портила состояние регистров;

А в случае с прерываниями у нас есть только один вариант решения, так как основной код программы не может знать, когда вызовется обработчик прерывания, а значит и сохранить нужные значения перед его вызовом не может.

Теперь, давайте представим, что обработчик прерывания написан так, чтобы сохранять значения всех регистров из r0-r31, которые он использует. Достаточно ли этого в примере с функцией __add16? Правильный ответ нет! Потому что состояние потока исполнения описывается не только регистрами общего назначения, но и регистром состояния SREG. Собственно, инструкция adc использует флаг C (Carry) регистра SREG, и если в обработчике прерывания выполнить действие, которое изменит флаг C, то функция __add16 опять может вернуть неверный результат. Таким образом, регистр SREG тоже необходимо сохранить.

Резюмируя все сказанное, начало обработчика прерывания таймера в нашем случае будет выгляедеть следующим образом:

```
1 timer0_compa:
2         push r24
3         push r25
4         push r26
5         push r27
6         push r16
7         push r18
8         in r18, SREG
```

Посмотрите, как мы поступаем в регистром SREG. Для него нет встроенного обозначения (как для регистров r0-r31) и его значение нельзя сохранить на стек используя push. Чтобы получить его значение, мы читаем из соответствующего адреса в пространстве ввода/вывода в регистр r18. В ходе обработки прерывания мы не будем изменять значение регистра r18, и потом просто восстановить SREG с помощью инструкции out из этого регистра.

Теперь посмотрим как будет выглядеть конец нашего обработчика прерываний, в котором мы восстановим все испорченные регистры:

```
1         out SREG, r18
2         pop r18
3         pop r16
4         pop r27
5         pop r26
6         pop r25
7         pop r24
8         reti
```

Отметим, что вместо обычного ret, для выхода из обработчика прерываний используется специальная инструкция reti. Необходимость специальной инструкции объясняется тем, что во время обработки прерываний может произойти другое прерывание, а процессор AVR не разрешает такой ситуации. Поэтому, перед тем как вызвать обработчик прерывания, процессор AVR сбрасывает бит I (Global Interrupt Enable) в регистре SREG. Если бит I сброшен - то все прерывания запрещены. Т. е. даже если таймер/счетчик зафиксировал совпадение и все его регистры настроены так, чтобы происходила прерывание его не произойдет, если флаг I сброшен. Короче говоря, флаг I - глобальный выключатель всех прерываний.

Возвращаясь к инструкции reti, ее отличие от инструкции ret заключается в том, что она не только достает со стека адрес возврата и передает по нему правление, но и устанавливает флаг I в регистре SREG, тем самым включая прерывания назад.

Кстати говоря, в наборе команд AVR есть две команды sei и cli, которые устанавливают и сбрасывают флаг I, соответственно. По-умолчанию, в начале программы флаг I сброшен. Это логично, потому что пока не настроен стек процессор не может вызывать прерывания - ему просто некуда складывать адрес возврата. Поэтому устанавливать флаг I нужно явно.

Наконец содержательная часть нашего обработчика прерывания таймера:

```
1         lds r24, JIFFIES
2         lds r25, JIFFIES + 1
3         lds r26, JIFFIES + 2
4         lds r27, JIFFIES + 3
5
6         clr r16
7         adiw r24, 0x0001
8         adc r26, r16
9         adc r27, r16
```

```
10
11          sts JIFFIES, r24
12          sts JIFFIES + 1, r25
13          sts JIFFIES + 2, r26
14          sts JIFFIES + 3, r27
```

Тут все довольно просто, сначала мы загружаем из памяти в регистры r24-r27 значения глобального счетчика. Затем прибавляем к нему 1. Код сложения немного отличается от известной вам функции __add32, но вы можете обратится за пояснениями в [AVR, b], неизвестными для вас являются только инструкция clr (которая очищает регистр, т. е. устанавливает его значение в 0) и инструкция adiw, которая складывает 16-битное значение в регистрах r24-r25 с непосредственным значением (от 0 до 63), в данном случае 1.

Заметим, что как аргумент adiw указывается только регистр r24 - это довольно распространенная ситуация[1], когда из пары регистров как операнд указывается только один, а второй подразумевается. В частности, в случае инструкции adiw, в качестве регистров операндов могут выступать только пары r24-r25, r26-r27, r28-r29, r30-r31, поэтому достаточно указать только первый регистр из пары, а второй определяется однозначно.

Теперь используя глобальный счетчик JIFFIES мы напишем функцию delay_ms, которая будет принимать в качестве параметра 16-битное значение в регистрах r16-r17, и выполняться примерно указанное число миллисекунд.

Идея работы функции довольно простая - читаем значение JIFFIES, прибавляем к нему значение в регистрах r16-r17. Далее в цикле читаем значение JIFFIES, и сравниваем его с полученной суммой. Если они не совпали повторяем тело цикла сначала, иначе завершаем функцию. Т. е. мы будем читать значение JIFFIES, пока оно не дойдет до нужного нам значения, как только это случилось мы завершаем функцию.

Теперь рассмотрим части функции подробнее, начиная с проверки входных значений:

```
1  delay_ms:
2           tst r16
3           brne start_delay
4           tst r17
5           brne start_delay
6           ret
```

[1] и не только для AVR

Этот код, проверяет равно ли значение в r16-r17 нулю. Если оно равно нулю, то мы выходим из функции. Зачем это нужно? Кроме того, что ожидать 0мс нет особого смысла есть и другая причина. Без этой проверки функция delay_ms может работать неправильно, если на вход передан 0. Представим, что сразу после того, как мы прочитали значение JIFFIES в delay_ms, произошло прерывание и значение JIFFIES было увеличено на 1. Управление возвращается назад в delay_ms. Мы увеличиваем прочитанное старое значение JIFFIES на 0 (нам передан 0 в качестве параметра), т. е. в регистрах мы просто храним старое значение JIFFIES, которое уже меньше чем актуальное значение JIFFIES. Теперь мы будем в цикле читать значение JIFFIES из памяти и сравнивать со старым значение JIFFIES - и они, очевидно, сравняются не раньше чем произойдет переполнение 32-битного значения JIFFIES.

Далее в коде идет сохранение всех используемых в функции регистров на стек, поэтому мы пропускаем эту неинтересную часть и переходим к первому чтению JIFFIES из памяти:

```
1         cli
2         lds r24, JIFFIES
3         lds r25, JIFFIES + 1
4         lds r26, JIFFIES + 2
5         lds r27, JIFFIES + 3
6         sei
```

Мы окружили чтение инструкциями cli и sei. Это нужно, чтобы значение JIFFIES не обновлялось пока мы его читаем, иначе мы можем прочтитать неправильное значение.

Далле мы прибавляем к прочитанному значению переданный параметр:

```
1         clr r18
2         add r24, r16
3         adc r25, r17
4         adc r26, r18
5         adc r27, r18
```

Тут нет ничего интересного, мы дополнили значение в регистрах r16-r17 до 32 бит используя регистр r18 с нулевым значением, в остальном это обычное сложение.

Теперь основной цикл ожидания:

```
1 test_again:
2         lds r31, JIFFIES + 3
3         lds r30, JIFFIES + 2
```

```
4           lds r29, JIFFIES + 1
5           lds r28, JIFFIES
6
7           cp r28, r24
8           brne test_again
9           cp r29, r25
10          brne test_again
11          cp r30, r26
12          brne test_again
13          cp r31, r27
14          brne test_again
```

Мы снова читаем значение JIFFIES, но уже в другие регистры и в другом порядке. На этот раз мы не запрещаем прерывания. Почему? Потому что мы не боимся прочитать неправильное значение. Чтобы не бояться прочитать неправильное значение, мы специально читаем байты от старшего к младшему. Почему это спасает нас от проблем? Нужно доказать два утверждения:

- функция delay_ms завершится не "слишком поздно", т. е. актуальное состояние глобального счетчика JIFFIES к моменту завершения будет "меньше или равно" значению JIFFIES, прочитанному в самом начале функции и увеличенному на переданный параметр;
- функция delay_ms завершится не "слишком рано", т. е. актуальное значение JIFFIES будет "больше либо равно" значению JIFFIES, прочитанному в самом начале функции и увеличенному на значение переданного параметра;

Отметьте, что "меньше или равно" и "больше либо равно" взяты в кавычки, так как при рассуждениях нужно учитывать переполнения, т. е. это не привычные отношения. Доказательство первого утверждения довольно очевидно, если вспомнить, что 1мс для 16МГц тактового генератора Arduino - это просто огромный интервал времени, за который цикл успеет проделать не одну итерацию, и хотя бы в одной из них будет считано правильное значение JIFFIES. Доказательство второй части несколько более трудоемкое, но все еще не сложное, достаточно рассмотреть, в каких ситуациях мы можем прочитать неправильное значение JIFFIES и показать, что прочитанное значение никак не может совпасть с нужным.

Конечно, чтобы не забивать себе голову такими деталями - гораздо проще отключить прерывания на время выполнения критических кода,

но, по возможности, стоит ограничивать время, которое прерывания были отключены. Поэтому я тут и расстарался, но вам совсем не обязательно повторять все эти сложности, пример специально усложнен, чтобы указать на проблемы, связанные с обработкой прерываний.

Оставшаяся часть фрагмента просто выполняет сравнение прочитанного значения JIFFIES с ожидаемым, и если хотя бы один из 4 байт не совпадает, то управление передается в начало цикла.

Осталась только часть функции, которая восстанавливает сохраненные на стеке регистры, но ее мы не будем детально расписывать, просто приведем всю функцию целиком:

```
1  delay_ms:
2          tst r16
3          brne start_delay
4          tst r17
5          brne start_delay
6          ret
7
8  start_delay:
9          push r16
10         push r17
11         push r18
12         push r24
13         push r25
14         push r26
15         push r27
16         push r28
17         push r29
18         push r30
19         push r31
20
21         cli
22         lds r24, JIFFIES
23         lds r25, JIFFIES + 1
24         lds r26, JIFFIES + 2
25         lds r27, JIFFIES + 3
26         sei
27
28         clr r18
29         add r24, r16
30         adc r25, r17
31         adc r26, r18
32         adc r27, r18
```

```
33
34 test_again:
35         lds r31, JIFFIES + 3
36         lds r30, JIFFIES + 2
37         lds r29, JIFFIES + 1
38         lds r28, JIFFIES
39
40         cp r28, r24
41         brne test_again
42         cp r29, r25
43         brne test_again
44         cp r30, r26
45         brne test_again
46         cp r31, r27
47         brne test_again
48
49         pop r31
50         pop r30
51         pop r29
52         pop r28
53         pop r27
54         pop r26
55         pop r25
56         pop r24
57         pop r18
58         pop r17
59         pop r16
60         ret
```

У нас почти все готово, осталась только основная программа. Она довольно очевидна. Начнем с заголовка:

```
1  #include "atmega328.h"
2
3          .device ATmega328P
4          .org 0x00
5          rjmp reset
6
7          .org 0x1c
8          rjmp timer0_compa
9
10         .org 0x34
11 reset:
```

Он должен быть вам хорошо знаком, за одним лишь исключением, мы добавили еще один rjmp по адресу 0x1c - это адрес нужного нам обработчика прерываний таймера/счетчика.

```
1  reset:
2          ldi r16, high(RAMEND)
3          out SPH, r16
4          ldi r16, low(RAMEND)
5          out SPL, r16
6
7          rcall setup_jiffies
8          sei
```

Следующая часть программы инициализирует стек, вызывает функцию настройки таймера setup_jiffies и включает прерывания.

```
1          ldi r18, bit(5)
2          clr r20
3          out DDRB, r18
4          ldi r16, 0xf4
5          ldi r17, 0x01
6  loop:
7          rcall delay_ms
8          eor r20, r18
9          out PORTB, r20
10         rjmp loop
```

Оставшаяся часть настраивает порт ввода/вывода, записывает в регистры r16-r17 значение 500 (задержка на 500мс), после чего в цикле вызывает delay_ms и изменят значение бита PORTB5 на противоположное (с учетом того, что все остальные биты имеют значение 0).

8 Самостоятельная работа: моргаем светодиодами правильно

Вы уже умеете моргать светодиодом, но пока вы моргаете светодиодом вы не можете делать ничего другого. В этом задании вы исправите это положение дел выполнив всю работу со светодиодом в обработчике прерывания. Кроме того, в этом задании вы измените форму сигнала.

Вам необходимо написать программу, которая тушит встроенный светодиод на 0.9 с, после чего зажигает его на 0.1 с и так по кругу.

Всю работу со светодиодом, кроме начальной инициализации вы должны выполнять в обработчике прерывания таймера.

Порядок выполнения работы:

1. Посчитайте, какое значение для сравнения нужно использовать, чтобы прерывание таймера генерировалось каждые 0.9 с (период - это величина обратно пропорциональная частоте)? Посчитайте это значение для всех возможных коэффициентов деления.

2. Посчитайте, какое значение для сравнения нужно использовать, чтобы прерывание генерировалось каждые 0.1 с? Посчитайте это значение для всех возможных коэффициентов деления.

3. Выберете наибольший коэффициент деления, при котором значения из пп. 1-2 будут целыми (или ближе всего к целым).

4. Выберите таймер/счетчик, регистр OCRxA которого сможет вместить наибольшее из посчитанных значений (будем использовать один таймер для отсчета обоих интервалов).

5. Настройте выбранный таймер/счетчик так, чтобы он генерировал исключение каждые 0.9 с. Обратите внимание на то, что разные таймеры/счетчики могут настраиваться по разному, а для 16 битного счетчика имеет значение в каком порядке вы записываете байты в 16 битные регистры, т. е. обратитесь к документации.

6. Реализуйте обработчик прерывания таймера таким образом, чтобы если в регистре TCNTx записано значение найденное в п. 1, то обработчик прерывания должен зажечь светодиод и записать в регистр OCRxA значение из п. 2. В противном случае потушите светодиод и запишите в регистр OCRxA значение из п. 1.

В итоге вы должны получить прогрмму, которая зажигает светодиод на 0.1 с, а затем тушит его на 0.9 с, итого светодиод будет моргать с частотой в 1 Гц.

Требования к решению:

1. я буду просматривать код программы, так что постарайтесь офрмить его аккуратно (расставьте отспуы, постарайтесь избавиться от магических чисел и прочее)

Для любителей поиграться со свободным временем: можете поиграться.

9 Последовательный интерфейс

В этой части мы увидим, как соединить наш AVR с другим устройством используя последовательный протокол передачи. Соединять мы будем Arduino с компьютером используя интерфейс[1] UART [Wikipedia, e] - один из множества последовательных интерфейсов передачи данных[2].

Мы не будем во всех деталях расписывать историю последовательных интерфейсов, что такое RS-232, SPI или I2C. Все эти вещи важно знать, т. к. SPI и I2C используются достаточно активно, но покрыть их все практическими примерами у меня просто нет возможности. Поэтому мы остановимся на самом простом варианте - UART.

Начнем мы с самых основ, почему интерфейс UART - последовательный интерфейс? Все очень просто, UART устроен так, что биты передаются друг за другом - последовательно. Это объединяет все последовательные интерфейсы.

Кроме того интерфейсы могут быть синхронными или асинхронными. Синхронные интерфейсы управляются тактовым генератором, в то время как асинхронным он не нужен. UART изначально асинхронный интерфейс, но вот расширение USART может работать и в синхронном режиме.

На какой скорости осуществляется передача данных в UART? Скорость может быть разной, главное, чтобы оба участника передачи использовали одну и ту же скорость. Скорость измеряется в бодах [Wikipedia, b]. Пока мы используем бинарное кодирование сигналов, скорость в 1 бод говорит, что мы можем передавать 1 бит в секунду[3]. Типичные скорости передачи 9600 бод, 19200 бод, 38400 бод, 57600 бод и 115200 бод[4].

Данные по UART передаются порциями[5] - мы будем называть их фреймами. Фреймы, опять же, могут быть разных размеров: 5 бит, 7 бит, 8 бит[6]. Наиболее часто используется порция в 8 бит - мы тоже будем

[1] даже не знаю, правильно ли называть это интерфейсом

[2] довольно старый, возможно, один из самых популярных

[3] бинарное кодирование в данном случае значит, что на уровне среды передачи есть всего два вида сигналов - ноль или единица, это не всегда правда и в этом случае 1 бод будет соответствовать другой скорости в битах

[4] это не единственные возможные скорости передачи, есть и другие

[5] а каждая порция последовательно побитово

[6] некоторые маргинальные реализации могут предоставлять и другие варианты

ее использовать. Порции данных передают от менее значимого бита к более значимому[1].

Помимо информационных битов в поток так же можно вставлять стартовый и стоповый биты. Пока по UART ничего не передается, напряжение в канале связи устанавливается в логическую 1, а стартовый бит - всегда логический 0. Таким образом при начале передачи фрейма напряжение переходит от логической 1 к логическому 0, это позволяет приемнику и передатчику синхронизироваться при начале передачи сообщения[2]. Стоповый бит наоборот соответствует логической единице.

Опционально вместе с данными может быть передан специальный бит четности - он нужен, чтобы проверить корректность полученных данных, и сейчас используется довольно редко. Итого типичный фрейм состоит из 1 стартового бита, 8 бит данных, 1 стопового бита - такой формат и будем использовать[3].

В контексте последовательных интерфейсов часто вспоминают еще и такое понятие как Flow Control. Это набор программных/аппаратных методов для управления скоростью передачи. Тут имеется ввиду количество переданных фреймов в единицу времени, а не скорость передачи одного фрейма (боды или baudrate). Baudrate всегда фиксирован, а вот фреймы можно передавать с разной частотой, и приемник может не справляться с потоком данных[4]. Но мы не будем касаться этой темы.

Пора перейти к практике. Как и с таймерами/счетчиками тут есть куча специальных регистров которые нужно конфигурировать. Кроме того, хитрые создатели Arduino еще и вставляют нам палки в колеса[5]. Делать будем программу не сильно более крутую, чем программа для моргания встроенным светодиодом. На этот раз мы будем читать один байт из последовательного интерфейса, и записывать его в PORTB, тем самым подавая сигналы сразу на несколько ног процессора[6].

Как и с таймерами/счетчиками, работать с UART можно с использованием прерываний или проверяя в цикле различные флаги. Как и в случае с таймерами мы будем использовать прерывания. Для нашей задачи в этом нет особых преимуществ и, в принципе, можно

[1] хотя для программиста это зачастую не важно

[2] а для асинхронного интерфейса это важно

[3] бывают и другие варианты, много других вариантов

[4] если каждый фрейм нуждается в какой-то очень трудоемкой обработке

[5] вообще говоря это не палки в колесах, а вполне разумное решение, но в нашем случае с ним придется немного побороться

[6] к которым, конечно, можно подключить светодиоды

было бы обойтись циклом с проверкой того, пришли ли данные или нет[1], но это слишком просто [2], а прерывания - супер важная концепция в современных архитектурах вычислительных систем.

Для начала разберемся, как настроить параметры UART. Нам нужно задать режим работы (синхронный/асинхронный, так как в AVR встроен именно USART контроллер) скорость работы последовательного интерфейса, установить используемый формат фрейма и включить прерывания. Сначала займемся первыми тремя пунктами.

Для передачи будем использовать символьную скорость в 9600 бод. Почему такую маленькую? Потому что для этой скорости ошибка будет меньше, и вероятность прочитать мусор из-за рассинхронизации будет меньше.

Для генерации сигналов UART на заданной частоте процессор AVR имеет baud rate генератор. Он чем то напоминает таймер/счетчик, только очень узко специализированный. В нем так же присутствует схема делитель частоты, которую необходимо настроить соответствующим образом.

Скорость определяется значениями регистров UBRR0L/UBRR0H, частотой тактового генератора, а кроме того режимом работы, в зависимости от которого частота baud rate генератора делится на 2, 8 или 16. Эти режимы в документации называются: synchronous master mode, asynchronous double speed и asynchronous normal, соответственно.

Нам подойдут только два последних[3]. Их этих двух выберем asynchronous double speed. Почему? Потому что звучит круто. Разумный выбор между ними должен основываться на допустимой величине ошибки. Если ожидается, что частота baud rate генератора приемника и передатчика могут сильно отличаться, то лучше не использовать double speed режим, так как он менее толернатен к ошибкам и может произойти рассинхронизация[4].

Чтобы посчитать необходимое значение регистров UBRR0L/UBBR0H используем формулу из документации[5]:

$$UBBR0 = \frac{F_{sys}}{8 \times BAUD} - 1 \tag{2}$$

где:

[1] это называется polling

[2] вы могли заметить, что я не ищу легких путей - я сам создаю себе трудности и с гордостью их преодолеваю

[3] потому что мы хотим использовать асинхронный режим работы

[4] диапазоны допустимых ошибок есть в документации

[5] впрочем подобную формулу вы уже видели раньше

- F_{sys} - частота тактовго генератора;
- $BAUD$ - требуемая скорость

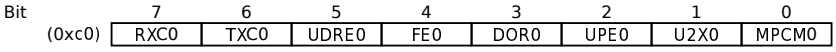

Figure 9.6: Регистр UCSR0A

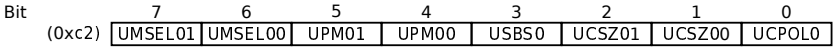

Figure 9.7: Регистр UCSR0C

В нашем случае получаем 207[1]. На самом деле в той же документации приведены таблицы с готовыми значениями и оценками погрешности, так что считать самостоятельно нужды не было. Очевидно, чем меньше оценка погрешности тем меньше вероятность рассинхронизации.

Теперь мы знаем, что нужно записать в UBRR0L/UBRR0H. Осталось одно маленькое замечание. Чтобы использовать double speed режим, нужно его включить, а для этого нужно записать в бит U2X (Double the USART Transmission Speed) регистра UCSR0A (USART Control and Status Register A fig. 9.6) единицу, а в биты UMSEL00-UMSEL01 (USART Mode Select) регистра UCSR0C (USART Control and Status Register C fig. 9.7) нули. Т. е. для выбора нужного нам режима и задания скорости работы необходимо заполнить следующие биты и регистры:

- UBRR0L - записать 207;
- UBRR0H - записать 0;
- бит U2X регистра UCSR0A - записать 1;
- биты UMSEL00-UMSEL01 регистра UCSR0C - записать 0;

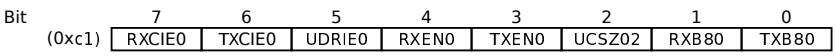

Figure 9.8: Регистр UCSR0B

Теперь разберемся с форматом фрейма. Тут создатели AVR предоставили целую пачку возможностей. Как я уже говорил мы

[1]число получилось нецелым - округляем к ближайшему целому

будем использовать следующий формат - один стартовый бит, один стоповый бит, 8 бит данных без проверки четности. За определение формата фрейма отвечают биты UCSZ00-UCSZ02 (USART Character Size), UPM00-UPM01 (USART Parity Mode) и USBS0 (USART Stop Bit Select) в регистрах UCSR0B (USART Control and Status Register B fig. 9.8) и UCSR0C.

Не будем вдаваться в подробности всех возможных комбинаций этих бит[1], а сразу перейдем к нужному нам варианту:

- UCSZ02 регистра UCSR0B - записать 0;
- биты UCSZ00-UCSZ01 регистра UCSR0C - записать 1;
- биты UPM00-UPM01 регистра UCSR0C - записать 0;
- бит USBS0 регистра UCSR0C - записать 0;

Теперь нужно включить UART приемник и включить прерывания при получении данных. Чтобы включить приемник нужно записать 1 в бит RXEN0 (Receiver Enable) регистра UCSR0B. Для включения прерываний при приеме данных необходимо записать 1 в бит RXCIE0 (RX Complete Interrupt Enable) того же регистра.

Остается самая малость - нужно понять, откуда собственно взять пришедшие данные? Все пришедшие данные хранятся в специальном регистра UDR0 (USART I/O Data Register)[2].

Начнем, собственно, писать код. Как всегда заведем макроопределения для всех используемых регистров и для отдельных битов в них:

```
1  #define UDR0        0xa6
2  #define UBRR0H      0xa5
3  #define UBRR0L      0xa4
4
5  #define UCSR0C      0xa2
6  #define UCPOL0      0x00
7  #define UCSZ00      0x01
8  #define UCSZ01      0x02
9  #define USBS0       0x03
10 #define UPM00       0x04
11 #define UPM01       0x05
12 #define UMSEL00     0x06
```

[1] потому что их 30 штук - все не перечислишь

[2] забавно, что это на самом деле не один регистра, два - для чтения и для записи

```
13 #define UMSEL01     0x07
14
15 #define UCSR0B      0xa1
16 #define TXB80       0x00
17 #define RXB80       0x01
18 #define UCSZ02      0x02
19 #define TXEN0       0x03
20 #define RXEN0       0x04
21 #define UDRIE0      0x05
22 #define TXCIE0      0x06
23 #define RXCIE0      0x07
24
25 #define UCSR0A      0xa0
26 #define MPCM0       0x00
27 #define U2X0        0x01
28 #define UPE0        0x02
29 #define DOR0        0x03
30 #define FE0         0x04
31 #define UDRE0       0x05
32 #define TXC0        0x06
33 #define RXCO        0x07
```

Теперь напишем функцию, которая настроит UART на прием данных и генерацию исключений:

```
1  setup_uart:
2          ldi r16, low(207)
3          sts iospace(UBRR0L), r16
4          ldi r16, high(207)
5          sts iospace(UBRR0H), r16
6
7          ldi r16, bit(U2X0)
8          sts iospace(UCSR0A), r16
9
10         ldi r16, (bit(UCSZ00) | bit(UCSZ01))
11         sts iospace(UCSR0C), r16
12
13         ldi r16, (bit(RXEN0) | bit(RXCIE0))
14         sts iospace(UCSR0B), r16
15         ret
```

Осталось написать обработчик прерывания и записать инструкцию rjmp в нужное место таблицы обработчиков прерываний:

```
1  uart_rx_complete:
```

```
2           push r16
3           lds r16, iospace(UDR0)
4           out PORTB, r16
5           pop r16
6           reti
```

Инструкция rjmp, согласно спецификации [AVR, a], должна располагаться по адресу 0x24 от начала, собственно:

```
1           .org 0x24
2           rjmp uart_rx_complete
```

Основная программа не сильно интересна, я приведу ее, но без пояснений:

```
1   reset:
2           ldi r16, high(RAMEND)
3           out SPH, r16
4           ldi r17, low(RAMEND)
5           out SPL, r16
6
7           ldi r16, 63
8           out DDRB, r16
9
10          rcall setup_uart
11          sei
12  loop:
13          rjmp loop
```

Теперь научимся передавать на Arduino что-нибудь. Для этого мы напишем скрипт на python[1]. Скрипту для работы нужна библиотека pyserial, которая "оборачивает" обычный posix интерфейс работы с файлами[2]. Выглядит скрипт не очень сложно:

```
1   #!/usr/bin/python
2   from contextlib import closing
3   import serial
4   import sys
5   import time
6
7   if __name__ == '__main__':
8       with closing(serial.Serial(sys.argv[1], baudrate=9600))
            ↪  as s:
```

[1] хватило бы и стандартных консольных утилит

[2] open/close/read/write/ioctl - все есть файл и тд, короче, вы должны быть в курсе

```
9           time.sleep(2)
10          s.write(bytearray([63]))
```

Он принимает на вход имя файла последовательного интерфейса. Это имя можно определить, воспользовавшись скриптом detect.sh, который мы вам уже показывали. Затем скрипт ждет[1] некоторое время, чтобы устройство загрузилось. Зачем ждать? Затем, что при открытии файла последовательного порта Arduino перезагружается. Тут нет никакой магии, просто они соединили пару контактов вместе, но мы не будем вдаваться в подробности. Зачем?

Дело в том, что чтобы мы могли заливать программу на устройство через удобный USB интерфейс, а не покупая[2] специальный программатор, на устройстве в специальной секции постоянной памяти содержится программа загрузчик. Эта программа получает управление, как только устройство запускается, после чего управление передается уже нашей программе.

Этот загрузчик ждет некоторое время[3] пока на UART не начнут приходить данные. Если данные так и не придут до истечения времени, то загрузчик завершается, а наша программа получает управление. Если же какие-то данные пришли, то загрузчик считает их новой программой, которую мы хотим записать в AVR и записывает их.

Так вот, если при открытии файла последовательного устройства сразу же перезапустить Arduino, то запустится этот загрузчик и начнет читать данные UART. Этим мы и пользуемся при загрузке программ на него. Что будет, если мы не перезапустим устройство? Загрузчик не получит управление и не запишет программу. Хорошо, но почему мы не можем сами нажать reset, чтобы перезапустить Arduino, когда загружаем на него новую программу? Вообще, мы можем, но придется аккуратно ловить момент, а с автоматической перезагрузкой при открытии соединения такого делать не нужно.

Но это было лирическое отступление о пользе и вреде загрузчиков. Вернемся к нашему таймауту. Мы ждем некоторое время, чтобы загрузчик убедился, что мы не собирается загружать на устройство новую программу и завершился передав управление нашей программе. AVR процессоры прекрасно могут работать и без загрузчика, но раз уж мы используем Arduino и у нас нет специального программатора, то нам придется мириться с таким поведением.

Что же мы пишем в последовательный интерфейс? Что значит число

[1] это как раз к вопросу о палках в колеса от создателей Arduino

[2] можно еще и самому сделать

[3] не знаю какое, но не долго

63? В числе 63 установлены все биты кроме 6 и 7. Почему все кроме них? Из 8 бит порта ввода/вывода B нам для общего использования доступны только 6. Еще две ноги порта B используется для подключения внешнего 16МГц тактового генератора.

В качестве резюме, могу обратить ваше внимание на то, что теперь мы можем с компьютера управлять зажиганием светодиодов - самое время сделать какую-нибудь прикольную гирлянду, описав в python логику и время зажигания/потухания светодиодов.

10 Самостоятельная работа: очень дорогой термометр

В этой самостоятельной работе вы подключите к Arduino аналоговый датчик темепературы и будете передавать его показания на компьютер по UART.

В качестве датчика температуры будем использовать LM335Z. LM335Z имеет три ноги:

- GND - должна быть подключена к GND (если смотреть на датчик со стороны его ног плоской частью наверх, то это самая правая нога);
- VCC - должна быть подключена к VCC через резистор (средняя нога);
- ADJ - используется для калибровки датчика (что важно в реальной жизни, и не очень важно для нас);

Показание датчика - это разность потенциалов между выходами GND и VCC. Чем больше напряжение, тем больше температруа. Нулевое напряжение соответсвует 0 градусов по Кельвину. Более формально, выходное напряжение примерно соответствует следующей формуле:

$$U = T * K$$

где, T - температура по Кельвину, а K - коэффициент, который в нашем случае равен примерно $10mV/K$, т. е. на каждый градус Кельвина напряжение изменяется на 10 mV.

Чтобы подключить этот датчик к Arduino, мы должны использовать один из его аналоговых портов (ищите надпись ANALOG IN на плате). С аналогового входа можно считать число от 0 до 1023 (включительно),

которое кодирует напряжение в диапазоне от GND до используемого референсного напряжения (bandgap volatage). Более фомально значение прочитанное с аналогового входа задается как:

$$\frac{ADC = V_{in} \times 1024}{V_{ref}}$$

где V_{in} - напряжение на входе, а V_{ref} - референсное напряжение.

Естественно аналогово-цифровой преобразовтель тоже нужно на строить для правильной работы. Настройка состоит из нескольких этапов:

1. выбор аналогового входа, с которого мы хотим снимать показания;
2. выбор источника референсного напряжения;
3. настройка делителя частоты преобразователя (преобразователь снимает среднее занчение на некотором интервале времени, поэтому и нужен правильно настроенный сигнал от таймера);
4. выбор режима работы (единоразовое измерение или постоянное измерение);

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	---	MUX3	MUX2	MUX1	MUX0

Figure 10.9: Регистр ADMUX

Начнем с начала, за выбор нужного аналогового входа отвечает регистр ADMUX fig. 10.9, точнее его биты MUX0-MUX3. Мы будем использовать 0 аналоговый вход, значит значение всех этих бит должно быть 0.

Выбор источника рефернсного напряжения определяется битами REFS0-REFS1. Мы будем использовать напряжение питания как источник, поэтому мы должны установить бит REFS0 в 1, а бит REFS1 оставить равным 0. Значение референсного напряжения предполагается равным 5 V[1].

За коэффициент деления частоты отвечают биты ADPS0-ADPS2 регистра ADCSRA fig. 10.10. Поддерживаются коэффициенты от 2 до 128. Согласно документации, аналогово-цифровому преобразователю

[1] что не совсем правда, или даже совсем не правда

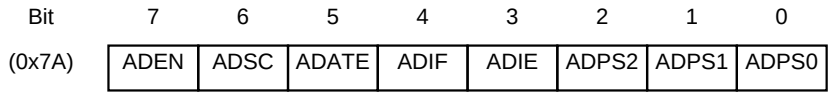

Figure 10.10: Регистр ADCSRA

треюуется частота от 50 кГц до 200 кГц. При чатстоте таймера в 16 МГц, нам подходит только один коэффициент деления из всех возможных - 128. Чтобы задать максимальный коэффициент деления нужно все биты ADPS0-ADPS2 установить в единицу.

Наконец, нам нужно выбрать режим работы, включить преобразователь и запустить его. Мы будем использовать преобразователь в так называемом free running режиме, т. е. преобразователь будет сам в цикле снимать показания с выбранного входа и сохранять результат в регистрах. За free running режим отвечает бит ADATE, т. е. чтобы включить этот режим, мы должны установить бит ADATE в единицу.

Чтобы включить преобразователь, нужно установить бит ADEN, а чтобы запустить преобразование нужно установить бит ADSC в регистре ADCSRA.

Осталось только узнать куда преобразователь будет сохранять результат? Для результата измерения выделены два специальных регистра ADCL и ADCH, которые находятся по адресам 0x78 и 0x79, соответсвенно. Обратите внимание, что важен порядок в котором вы читаете значения из этих регистров: первым вы должны прочитать значение из ADCL, и только затем прочитать значение из ADCH. Причем, если вы прочитали значение из ADCL, то вы обязательно должны прочитать значение из ADCH, в противном случае преобразователь не сможет записать в регистры новое измеренное значение[1].

Порядок выполнения задания. На этот раз разделим задание на несколько частей. Часть первая - сборка схемы.

Подключите температурный датчик к Arduino, как это показано на fig. 10.11. При подключении, пожалуйста, не отрывайте ему ноги - он этого не любит.

Часть вторая - инициализация аналогово-цифрового преобразователя. Для этого проделайте все шаги по списку:

[1]на самом деле первое чтение из ADCL блокирует регистры до тех пор, пока вы не прочитаете что-то из ADCH, таким образом преобразователь не сможет параллельно с вами писать в эти регистры и вы не прочитаете из них мусор

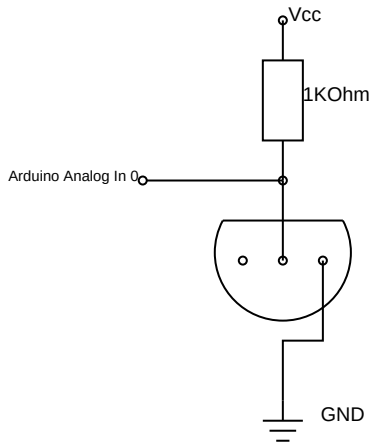

Figure 10.11: Схема подключения LM335Z

1. выберите через регистр ADMUX нужный аналоговый вход и источник референсного напряжения;

2. задайте через регистр ADCSRA нужный коэффициент деления частоты и установите биты ADATE, ADEN и ADSC, чтобы выбрать free running режим, включить преобразователь и запустить измерение соответственно.

Часть третья - инициализация UART. Во многом она такая же как и в примере, но на этот раз мы будем передавать данные, а не получать, и для простоты не будем использовать прерывания. Для инициализации проделайте все шаги по списку:

1. задайте скорость в 9600 бод в Asynchronous double speed режиме (как было показано в примере);

2. установите формат фрейма - 8 бит данных, 1 стартовый бит и 1 стоповый бит (опять же как в примере);

3. включите UART передатчик (бит TXEN0 в UCSR0B), но не включайте генерацию прерываний;

Наконец, часть третья - заключительная. Вы должны создать бесконечный цикл, а тело цикла последовательно выполняет следующие действия:

1. считываем результаты измерения в регистры (результаты хранятся в ADCL и ADCH);

2. передаем младший байт результата (записываем младший байт в регистр UDR0);
3. дожидаемся завершения передачи (ждем, пока флаг UDRE регистра UCSR0A не установится в 1);
4. передаем старший байт результата;
5. дожидаемся завершения передачи;

Требования к реализации: Код этого задания я не буду даже пытаться смотреть, достаточно убедить меня, что все работает. Для этого будет полезно (читай обязательно) иметь скрипт, который читатет данные от Arduino и печатает их в консоль.

Так как температура в пощенеии примерно постоянная, то будет полезно искусственно изменить температуру датчика. Поднять температуру датчика не трудно своим телом, короче говоря, сожмите его пальцами.

References

[AVR, a] *8-bit Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*. Atmel. `http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-3 datasheet_complete.pdf`.

[AVR, b] *AVR Instruction Set Manual*. Atmel. `http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf`.

[Wikipedia, a] Wikipedia. Atmel avr. `https://en.wikipedia.org/wiki/Atmel_AVR`. [Online; accessed 22-November-2014].

[Wikipedia, b] Wikipedia. Baud. `https://en.wikipedia.org/wiki/Baud`. [Online; accessed 7-December-2014].

[Wikipedia, c] Wikipedia. Dual in-line package. `https://en.wikipedia.org/wiki/Dual_in-line_package`. [Online; accessed 22-November-2014].

[Wikipedia, d] Wikipedia. Intel hex. `https://en.wikipedia.org/wiki/Intel_HEX`. [Online; accessed 22-November-2014].

[Wikipedia, e] Wikipedia. Universal asynchronous receiver/transmitter. `https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter`. [Online; accessed 7-December-2014].