Программирование на необитаемом острове

Микоян Филипп

Московский физико-технический институт (национальный исследовательский университет)

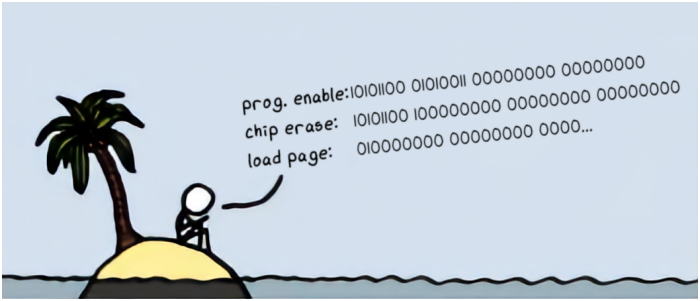


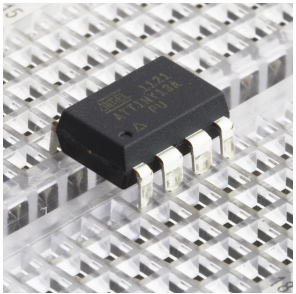
Рисунок 1 – Вольная интерпретация комикса xkcd

Рисунок 2 – Итак, ваш самолёт разбился...

Что у нас есть в наличии?

- Дикие лимоны
- Самолётная фара и высоковольтный аккумулятор для неё (в нашем случае лампочка накаливания)
- Спичечный коробок с:
  - Микроконтроллером AVR Attiny13
  - Несколькими проводами
  - Парой кнопок

Попробуем собрать из этого аварийный маяк, мигающий с периодом 1 секунда.



(a) Микроконтроллер ATtiny13A

(b) Релейный модуль для макетных плат

Рисунок 4 – Лимонная кислота — электролит. С каждого лимона получаем почти 1В

План действий

- Написать на Паскале программу, которая сможет ждать полсекунды.
- Переписать её на ассемблер.
- Добавить в программу мигание светодиодом.
- Понять, как она представляется в машинных кодах.
- Прочитать, какими командами её можно записать в память микроконтроллера.
- Изучить протокол `SPI` передачи данных.
- Прошить микроконтроллер.

Как микроконтроллеру ждать полсекунды?

Микроконтроллер 'ждёт', исполняя инструкции. Исполнение одной инструкции занимает какое-то время, определяемое частотой работы микроконтроллера(тактовой частотой) и числом тактов, необходимых на исполнение инструкции. Имеем систему:

$$\begin{cases} f = 128\ \text{КГц} = 128\ 000 \text{ тактов в секунду} & \text{— тактовая частота микроконтроллера} \\ T/2 = 0.5 \text{ секунды} & \text{— необходимое время ожидания} \\ N = (T/2) \cdot f = 64\ 000 & \text{— тактов нужно подождать} \end{cases}$$

Вопрос: а какова тактовая частота процессора в вашем персональном компьютере?

За сколько тактов процессора исполняется данная программа?

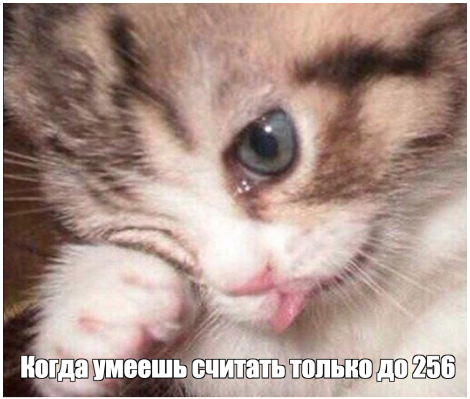
```
// Эта программа уменьшает значение
// переменной 'a' от десяти до нуля,
// т.е. считает до десяти.
var
    a: integer;

begin
    a := 10;
    repeat
        a := a - 1;
    until a = 0;
end.
```

```
// Та же самая программа,
// но использует goto и dec
var
    a: integer;

label
    loop;

begin
    a := 10;
loop:
    dec(a); // идентично a = a - 1;
            // Dec = Decrement = уменьшить на 1
    if (NOT (a = 0)) then
        goto loop;
end.
```

В процессоре нашего микроконтроллера `dec()` выполняется ровно за один такт процессора. Проверка условия в `if` и `goto` тоже тратят на исполнение по одному такту. Таким образом, для исполнения данного цикла с N итерациями требуется $3 * N$ тактов. В данном случае 30.

Сколько нужно регистров, чтобы написать такой длинный цикл?

Итак, чтобы подождать полсекунды, нужно прокрутить $64\ 000$ тактов процессора, а так как каждая итерация исполняется за 3 такта, цикл придётся написать на $\frac{64\ 000}{3} \approx 21500$ итераций. Проблема в том, что в микроконтроллере переменные процессора — регистры — занимают всего один байт и умеют считать только до 256. Если же их два, то уже можно создать цикл из $256 * 256 = 65536$ итераций.



Как посчитать до 21 500 с двумя байтами?

Программа ниже, имея в распоряжении два байта, считает до двадцати одной тысячи. Она уменьшает `r2` от 255 до 0, и, когда эта переменная становится равной нулю, уменьшает `r1` на единичку.

Таким образом, каждое уменьшение `r1` — это 256 проходов внутреннего цикла. В сумме до обнуления `r1` исполняется $\approx 84 * 256 \approx 21500$ итераций внутреннего цикла, на что затрачивается $21\,500 \cdot 3 = 64\,500$ тактов и что создаёт задержку приблизительно в $\frac{64500}{128000} \approx 0.5$ секунды. Когда `r1` обнуляется, программа заканчивает исполнение.

```
var r1, r2: byte;

label loop;

begin
    r1 := 83;   // r1 - 'старший разряд', уменьшается в 256 раз реже, чем r2
    r2 := 255; // r2 - 'младший разряд', постоянно бегает от 255 до 0

loop:
    dec(r2);
    if (NOT (r2 = 0)) then
        goto loop;
    dec(r1);
    if (NOT (r1 = 0)) then
        goto loop;
end.
```

Для начала перепишем простой цикл на ассемблер

```
var
    a: integer;

label
    loop;

begin
    a := 10;
loop:
    dec(a);
    if (NOT (a = 0)) then
        goto loop;
end.
```

```
;    Комментарии на ассемблере пишутся так.

;    В этой программе роль переменной
; 'a' играет регистр 'r20'.

    ldi r20, 10 ; r20 := 10;
                ; ldi = LoaD Immediate =
                ; = 'положить число в регистр'
loop:
    dec  r20    ;    Вычитает из значения регистра
                ; единичку и присваивает ему.
                ;    Если значение стало равно нулю,
                ; 'выставляет' определённый флаг
                ; Z (Zero) в состоянии процессора.

    brne loop   ;    brne = BRanch Not Equal = прыгни
                ; на метку, если не равно (нулю).
                ;    Смотрит, выставлен ли флаг Z.
                ; Если не выставлен, прыгает на метку,
                ; иначе ничего не делает.
```

Теперь посчитаем до двадцати тысяч на ассемблере

```
var r19, r20: byte;

label loop;

begin
    r19 := 83;   // старший разряд
    r20 := 255; // младший разряд

loop:
    dec(r20);
    if (NOT (r20 = 0)) then
        goto loop;
    dec(r19);
    if (NOT (r19 = 0)) then
        goto loop;
end.
```

```
        ldi   r19, 83   ; r19 := 83;
        ldi   r20, 255  ; r20 := 255;
loop:
        dec   r20
        brne  loop
        dec   r19
        brne  loop
```

Научимся менять напряжение на ножке

Выставление напряжения на ножке делается очень просто, оно почти ничем не отличается от присваивания значения регистру. Микроконтроллер ATtiny13 имеет два особых регистра — `DDRB` и `PORTB`. Эти регистры — битовые маски, т.е. каждому биту в этих регистрах соответствует своя ножка.
Например, присвоив число $2 = 0b10 = (1 << 1)$, регистру `DDRB`, можно настроить ножку номер 1 как `OUTPUT` — выходную. А присвоив число 2 регистру `PORTB`, можно выставить на первой ножке напряжение 5 В.

Код для Ардуино на языке C++

```
// Настраивает первую ножку на выход
// (это важно, так как ножки могут
// также работать на вход)
  pinMode(1, OUTPUT);

// Выставляет единицу на ножке
  digitalWrite(1, HIGH);

// Вечный цикл
  while (1);
```

Тот же код на ассемблере

```
; Инструкция out разрешает только
; присваивания между регистрами,
; так что нужно сначала присвоить
; число обычному регистру, и только
; потом присвоить особому регистру
; этот обычный.
    ldi r24, 0x02
; out присваивает регистрам DDRB и
; PORTB значение регистра r24, т.е. 0x02
    out DDRB, r24
    out PORTB, r24
; rjmp - аналог goto
; Эта строка эквивалентна вечному циклу
  L1: rjmp L1
```

Будем очень часто менять напряжение на ножке

```
    ldi r24, 0x02   ; r24 := 2
    ldi r25, 0x00   ; r25 := 0
    out DDRB, r24   ; DDRB := r24 := 2, настроить
                    ; первую ножку как OUTPUT
L1:
    out PORTB, r24  ; PORTB := r24 := 2, выставить
                    ; напряжение 5В на первой ножке
    out PORTB, r25  ; PORTB := r25 := 0, убрать
                    ; напряжение с первой ножки
    rjmp L1         ; goto L1
```

Теперь будем менять напряжение на ножке раз в полсекунды

```
    ; Исходная настройка
    ldi r24, 0x02
    ldi r25, 0x00
    out DDRB,  r24

    ; Главный цикл
L1:
    out PORTB, r24
    rcall WAIT      ; Вызывает функцию WAIT()
    out PORTB, r25
    rcall WAIT      ; Вызывает функцию WAIT()
    rjmp  L1

    ; Функция WAIT()
WAIT:
    ldi  r19, 83   ; r19 := 83;
    ldi  r20, 255  ; r20 := 255;
loop:
    dec  r20
    brne loop
    dec  r19
    brne loop
    ret  ; ret = RETurn. Прыгает туда, где был сделан rcall
```

Как происходит загрузка программы в микроконтроллер?

1. Передаётся последовательность байт `Programming Enable` — начать программирование.
2. Если необходимо, передаётся команда `Chip Erase` — очистить память микроконтроллера.
3. Запись программы в память происходит постранично. Страница — последовательность из 16 инструкций. Каждая инструкция занимает два байта, т.е. размер страницы равен `32` байтам, каждый из которых записывается отдельной командой `Load Program Memory Page`.
4. После заполнения страницы она записывается в память командой `Write Program Memory Page`.

Table 17-9. Serial Programming Instruction Set

Instruction	Instruction Format				Operation
	Byte 1	Byte 2	Byte 3	Byte4	
Programming Enable	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx	Enable Serial Programming after RESET goes low.
Chip Erase	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx	Chip Erase EEPROM and Flash.
Read Program Memory	0010 H000	0000 000**a**	**bbbb bbbb**	**oooo oooo**	Read **H** (high or low) data **o** from Program memory at word address **a:b**.
Load Program Memory Page	0100 H000	000x xxxx	xxxx **bbbb**	**iiii iiii**	Write **H** (high or low) data **i** to Program memory page at word address **b**. Data low byte must be loaded before Data high byte is applied within the same address.
Write Program Memory Page	0100 1100	0000 000**a**	**bbbb** xxxx	xxxx xxxx	Write Program memory Page at address **a:b**.

Рисунок 5 – Скриншот спецификации Atmega13

Представление инструкции в машинных кодах

Подробно разребрём процесс записи в память микроконтроллера следующей программы из одной инструкции:

```
ldi r19, 0x29
```

Для начала поймём, что именно мы хотим написать в память. Для этого откроем спецификацию языка ассемблера для нашего микроконтроллера:

LDI - Load Immediate

Description:

Loads an 8 bit constant directly to register 16 to 31.

Operation:

(i)Rd ← K

Syntax: Operands:

(i) LDI Rd,K 16 ≤ d ≤ 31, 0 ≤ K ≤ 255

16-bit Opcode:

1110	KKKK	dddd	KKKK

В спецификации данной инструкции, во-первых, написано, что она делает — загружает восьмибитную константу в регистр с номером от 16 до 31. То, что нужно, для регистра r19.

Во-вторых, написано, что в качестве аргумента она принимает число d: $16 \leq d \leq 31$ — номер регистра, младшие четыре бита которого хранятся во втором октете двухбайтной инструкции. Ещё одним аргументом является число K: $0 \leq K \leq 255$, которое будет помещено в регистр. Его младшие четыре бита располагаются в первом октете инструкции, а старшие четыре бита в третьем октете. Четвёртый же октет всегда равен $0b1110$.

Используя эту информацию, получим, каким двум байтам соответствует наша инструкция: $1110\ 0010\ 0011\ 1001_2 = E239_{16}$

Загрузка инструкции в память микроконтроллера

Итак, мы получили, что нашей инструкции

```
ldi r19, 0x29
```

соответствует пара байт $E239_{16}$. Так как архитектура AVR — little-endian, она хранит 'младшие' байты по младшим адресам, и в память нужно поместить выражение $39E2_{16}$, т.е. с поменянными местами байтами. Повсеместно используемые архитектуры компании Intel `i386` и `x86_64` тоже являются little-endian и хранят данные таким же образом.
Используя информацию с последних трёх слайдов, запишем последовательность байт, которую нужно подать на микроконтроллер для записи в него нашей программы:

```
10101100 01010011 00000000 00000000 // начать программирование
10101100 10000000 00000000 00000000 // очистить память

  номер байта       номер инструкции
внутри инструкции   внутри страницы
      V                 VVVV
01000000 00000000 00000000 00111001 // загрузить младший байт 0x39 в инструкцию номер 0
01001000 00000000 00000000 11100010 // загрузить старший байт 0xE2 в инструкцию номер 0
                           ^^^^^^^^
                           сам байт
                 номер страницы
                    V VVVV
01001100 00000000 00000000 00000000 // записать заполненное выше в страницу номер 0
```

Что записывать — понятно; а теперь — как записывать?

Самый распространённый и простой протокол для прошивки микроконтроллеров AVR — SPI(Serial Peripheral Interface) — последовательный периферийный интерфейс. Мы будем использовать следующие три провода:

- RESET — для разрешения прошивки нужно подать на этот провод 0.
- SCK — тактовый сигнал, который синхронизирует передачу данных.
- MOSI — провод, по которому передаются сами данные.

На провод SCK подаётся через равные промежутки времени ноль и единица. Частота смена единицы на ноль и обратно определяет тактовую частоту шины, определяющую скорость передачи данных. На каждый такт передаётся один бит информации — приёмник считывает значение на проводе MOSI в момент, когда значение на SCK меняется с нуля на единицу.

Рисунок 6 – Скриншот передачи байта 0b10100011

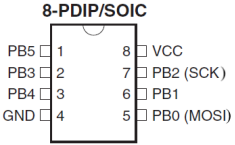


Рисунок 7 – Назначение ножек ATtiny13

Приступим к сборке схемы?

Простой расчёт показывает, что программа для нашего маяка, состоящая из пятнадцати инструкций, занимает $(15 \text{ инструкций } * 2 \frac{\text{байт}}{\text{инструкция}}) = 30$ байт. А так как для записи каждого байта приходится вводить ещё три байта адресации, да ещё 12 байт уйдёт на команды включения программирования, очищения памяти и записи страницы, в сумме получается $(30 * 4 + 12) = 132$ байта $= 1056$ бит $= 1056$ нажатий на одну только тактовую кнопку. Времени на это может хватить только на необитаемом острове, к тому же вероятность ошибки очень велика.

Также при сборке схем с кнопками типичной, но сложноустранимой проблемой является дребезг контактов — явление, возникающее потому, что в момент нажатия на кнопку контакты в ней из-за упругости несколько раз соударяются друг с другом, что приводит к многочисленным, неконтролируемым и очень коротким скачкам напряжения. Несмотря на их длительность, даже один скачок может испортить весь процесс прошивки.

Мы не будем углубляться в теорию методов борьбы с дребезгом — на необитаемом острове можно будет опускать два провода в воду, замыкая и размыкая цепь. А на материке мы воспользуемся самосборной схемой, позволяющей 'зашивать' программу по байту за раз. На работу же протокола `SPI` мы посмотрим с помощью логического анализатора.

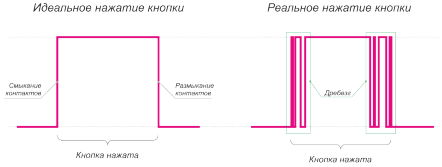


Рисунок 8 – Дребезг контактов

Как это — прошивать по байту за раз?

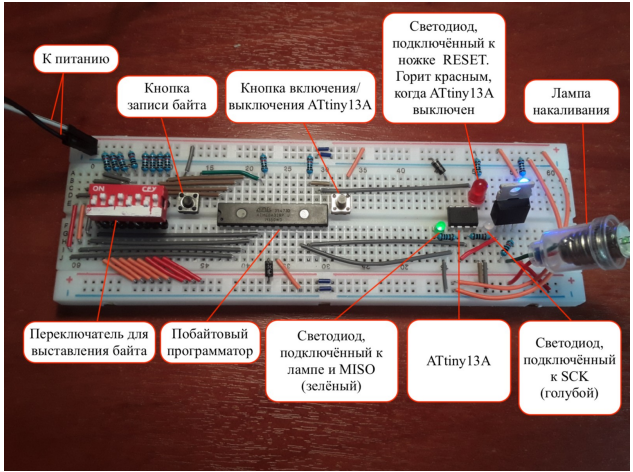


Рисунок 9 – Схема для побайтовой прошивки ATtiny13

- Переключатель. Ползунки на нём определяют значение байта, который будет отправлен на ATtiny13.
- Кнопка записи. При нажатии на неё состояния ползунков на переключателях считываются прошивателем и отправляются по протоколу SPI на ATtiny13.
- Кнопка RESET. При нажатии на эту кнопку на ножке RESET ATtiny13 выставляется нулевое напряжение, чтобы разрешить программирование. При этом загорается красный светодиод.
- Прошиватель — микросхема, реализующая всю логику, описанную выше.
- ATtiny13A. Прошиваемый микроконтроллер.
- Светодиоды MOSI и SCK(голубые). Часто мигают, когда происходит передача данных по SPI.
- Светодиод RESET(зелёный). Горит, когда на ножке RESET низкое напряжение, т.е. когда возможно программирование.
- Светодиод выходной(жёлтый). Горит в соответствии с программой на ATtiny13, используется при отладке в отсутствие аварийной лампы.

Как ещё более ускорить процесс прошивки?

Даже с побайтовым программатором пришлось бы выставлять значения 132^{x} байт, поэтому, чтобы ещё более ускорить процесс прошивки без потери наглядности, было решено предзагрузить в память программу, а вручную переписать лишь одну инструкцию, определяющую частоту мерцания маяка.

Так как запись в память микроконтроллера возможна только целой страницей, пришлось немного 'разнести' программу по памяти и добавить ещё одну инструкцию `rjmp(goto)`, которая 'перепрыгивает' мусор, находящийся в конце страницы №1. Переписывать в приведённой ниже программе мы будем только страницу №1.

Страница №0

```
; Исходная настройка
    ldi r24, 0x02
    ldi r25, 0x00
    out DDRB, r24

; Главный цикл
L1:
    out PORTB, r24
    rcall WAIT
    out PORTB, r25
    rcall WAIT
    rjmp L1
```

Страница №1

```
; Начало функции WAIT()
WAIT:
    ldi r19, 0x53 ; 83d
    rjmp wait2
```

Страница №2

```
; Продолжение функции WAIT()
wait2:
    ldi r20, 255
loop:
    dec r20
    brne loop
    dec r19
    brne loop
    ret ; конец функции wait
```

На что будем переписывать страницу №1?

Инструкции в старой версии страницы №1:

```
ldi   r19, 0x53  ; 0x53 = 83d
rjmp  wait2      ; rjmp — это относительный прыжок. Расстояние до начала след. страницы 14 байт
```

Инструкции в новой версии страницы №1:

```
ldi   r19, 0x29  ; 0x29 = 41d, в три раза меньше, чем 83d
rjmp  wait2
```

Description:

Relative jump to an address

Operation:

(i)PC ← PC + k + 1

16-bit Opcode:

1100	kkkk	kkkk	kkkk

Рисунок 10 – Выдержка из спецификации ассемблера AVR

Для приведённой инструкции `ldi` мы уже получали представление в машинных кодах на предыдущих слайдах: `0x39E2`.

Инструкция `rjmp`, аналог `goto` из Паскаля, расшифровывается как Relative JuMP — относительный прыжок. Это значит, что если аргумент `rjmp`, например, равен $14_{10} = E_{16}$, то управление перенесётся на 15 инструкций вперёд. Об этом написано в спецификации: управление передаётся на инструкцию $k+1$ относительно текущей, где k — аргумент `rjmp`.

Как видно из спецификации, первые три октета занимает аргумент. Это значит, что в память(с учётом little-endian) нужно будет написать `0x0EC0`.

Последовательность байт для прошивки

Итак, в память микроконтроллера нужно записать следующую последовательность: `0x39E20EC0`.
В соответствии с алгоритмом загрузки программ в память запишем последовательность команд для перезаписи страницы №1. Обратите внимание, что в этом случае команда `chip erase` не подаётся, так как мы хотим переписать только одну страницу.

```
    10101100 01010011 00000000 00000000 // начать программирование

  номер байта       номер инструкции
внутри инструкции   внутри страницы
      V                 VVVV
   01000000 00000000 00000000 00111001 // загрузить младший байт 0x39 в инструкцию номер 0
   01001000 00000000 00000000 11100010 // загрузить старший байт 0xE2 в инструкцию номер 0

   01000000 00000000 00000001 00001110 // загрузить младший байт 0x0E в инструкцию номер 1
   01001000 00000000 00000001 11000000 // загрузить старший байт 0xC0 в инструкцию номер 1
                              ^^^^^^^^
                              сам байт
                  номер страницы
                     V VVVV
   01001100 00000000 00010000 00000000 // записать заполненное выше в страницу номер 1
```

Всего для перепрошивки микроконтроллера требуется передать на него 24 байта. С этой задачей, благодаря побайтовой записи, можно справиться за пару минут.

Как прошивать с помощью побайтового программатора?

1. Вставить белый провод от шнура USB в отверстие рядом с красной дорожкой, а чёрный — в отверстие рядом с синей.
2. Вставить шнур USB в адаптер питания, а тот — в розетку.
3. Если зелёный светодиод мигает, понаблюдать за ним и на глаз оценить частоту мигания. Он должен мигать с периодом примерно одна секунда. Если это не так, нужно одновременно нажать на обе кнопки.
4. Чтобы начать программировать микроконтроллер, для начала нужно нажать на высокую кнопку подальше от переключателей — она выставит низкое напряжение на ножке `RESET` нашего ATtiny13. Загорится красный светодиод. Это значит, можно начинать программирование.
5. Чтобы "отправить" на ATtiny13 один байт, нужно выставить его на переключателях(младший бит справа, старший слева), затем (при горящем красном светодиоде) нажать на низкую кнопку рядом с переключателем. Синяя кнопка мигнёт — это значит, что байт успешно отправлен.
6. Если в ходе записи совершена ошибка, нажмите два раза на кнопку `RESET`, чтобы перезагрузить микроконтроллер и сбросить записанное.
7. После окончания программирования нужно нажать на кнопку `RESET`, чтобы запустить перепрограммированный микроконтроллер.

Проверим результат нашей работы

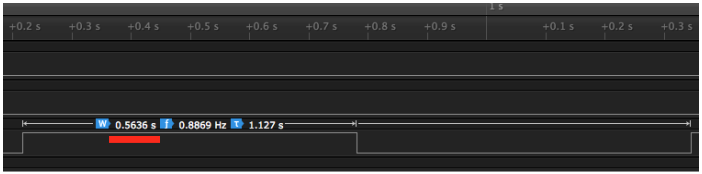


Рисунок 11 – Выходной сигнал при исполнении предзагруженной программы. Длина импульсов 0.56 отличается от половины секунды вследствие неточных расчётов и колебаний тактовой частоты.

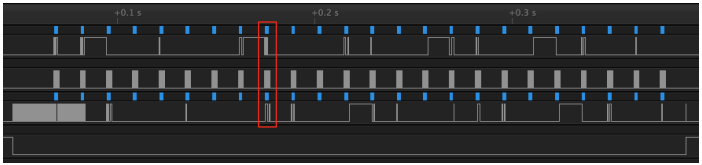


Рисунок 12 – Процесс переписывания страницы №1 микроконтроллера. Каждая синяя полоса — передача одного байта. Очень частые штрихи слева снизу, которые выглядят как сплошная полоса — это мусорный выход с ножки микроконтроллера, когда он уже перезагружен(на ножке `RESET`), но команда `Programming Enable` ещё не подана.

Проверим результат нашей работы

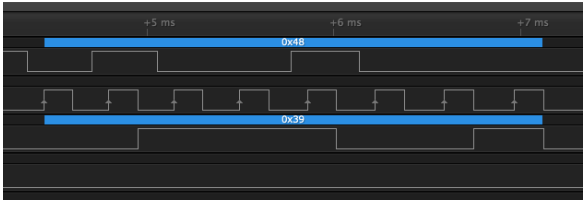


Рисунок 13 – Передача одного байта в ходе переписывания страницы №1 микроконтроллера. На этом изображении показана часть предыдущего снимка экрана, отмеченная красным прямоугольником. Этот байт, `0x48` — начало загрузки старшего байта инструкции `ldi r19, 0x29`.

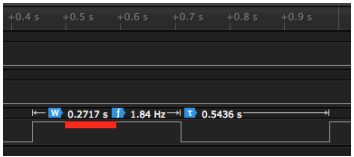


Рисунок 14 – Выходной сигнал при исполнении модифицированной программы. Длина импульсов `0.27` отличается от `0.56`, как и рассчитывалось, почти в два раза.

Напоследок взглянем, как представляется вся программа в машинных кодах

Страница №0

```
; Исходная настройка
      ldi r24, 0x02   ; 0x82E0
      ldi r25, 0x00   ; 0x90E0
      out DDRB, r24   ; 0x87BB

; Главный цикл
L1:
      out PORTB, r24  ; 0x88BB
      rcall WAIT      ; 0x0BD0
      out PORTB, r25  ; 0x98BB
      rcall WAIT      ; 0x09D0
      rjmp L1         ; 0xFBCF
```

Страница №1

```
; Начало функции WAIT()
WAIT:
      ldi r19, 0x53   ; 0x33E5
      rjmp wait2      ; 0x0EC0
```

Страница №2

```
; Продолжение функции WAIT()
wait2:
      ldi r20, 0xFF   ; 0x4FEF
loop:
      dec r20         ; 0x4A95
      brne loop       ; 0xF1F7
      dec r19         ; 0x3A95
      brne loop       ; 0xE1F7
      ret ; конец функции wait
```

Опционально: разберём, почему, например, инструкция `out DDRB, r24` представляется в машинных кодах именно так.

Чему мы научились?

- Считать до двадцати тысяч, имея только однобайтные регистры.
- Считать до двадцати тысяч на ассемблере.
- Мигать светодиодом/аварийным маяком каждые полсекунды на ассемблере.
- Переписывать программу с языка ассемблера на машинные коды.
- Загружать машинные коды по `SPI` в память микроконтроллера.

Поскольку у нас осталось свободное время, 'прошьём' микроконтроллер программой, которая обсуждалась в начале лекции:

Мигаем светодиодом так часто, как это возможно

```
    ldi r24, 0x02   ; 0x82E0
    ldi r25, 0x00   ; 0x90E0
    out DDRB,  r24  ; 0x87BB
L1:
    out PORTB, r24  ; 0x88BB
    out PORTB, r25  ; 0x98BB
    rjmp  L1        ; 0xFDCF
```

Эта программа будет мигать светодиодом с частотой около 32 кГц, так как тактовая частота микроконтроллера равна 128 кГц, а исполнение одной итерации цикла занимает четыре такта(по одному на каждый из `out`-ов и ещё два такта на `rjmp`).

Таких частых миганий светодиода мы точно не увидим глазом, разве что заметим уменьшение его яркости вследствие того, что часть времени он находится в выключенном состоянии. Поэтому результат загрузки данной программы придётся смотреть на логическом анализаторе.

Частые мигания: прошивка

```
10101100  01010011  00000000  00000000 // program enable
10101100  10000000  00000000  00000000 // chip erase
// ldi r24, 0x02
01000000  00000000  00000000  10000010 // load addr.0000  low byte 82
01001000  00000000  00000000  11100000 // load addr.0000 high byte E0
// ldi r25, 0x00
01000000  00000000  00000001  10010000 // load addr.0001  low byte 90
01001000  00000000  00000001  11100000 // load addr.0001 high byte E0
// out 0x17(DDRB), r24
01000000  00000000  00000010  10000111 // load addr.0010  low byte 87
01001000  00000000  00000010  10111011 // load addr.0010 high byte BB
// L1: out 0x18(PORTB), r24
01000000  00000000  00000011  10001000 // load addr.0011  low byte 88
01001000  00000000  00000011  10111011 // load addr.0011 high byte BB
// out 0x18(PORTB), r25
01000000  00000000  00000100  10011000 // load addr.0100  low byte 98
01001000  00000000  00000100  10111011 // load addr.0100 high byte BB
// rjmp L1
01000000  00000000  00000101  11111101 // load addr.0101  low byte FD
01001000  00000000  00000101  11001111 // load addr.0101 high byte CF

01001100  00000000  00000000  00000000 // write page 0
```

На изображении ниже представлен выходной сигнал при исполнении программы, мигающей светодиодом в бесконечном цикле. Частота мигания 28 кГц почти не отличается от предсказанной. Напряжение на ножке нулевое $\approx \frac{26}{35} \approx 75\%$ времени. Это связано с тем, что инструкция `rjmp` тратит на исполнение два цикла, а при её исполнении светодиод остаётся включённым. Таким образом, три цикла из четырёх напряжение на ножке положительное, что отлично согласуется с измерениями.

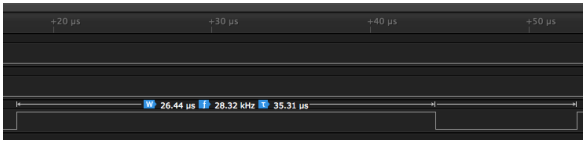


Рисунок 15 – Выходной сигнал часто мигающей программы