Cookbook for Assembly Programming with Arduino and plain 8bit Atmel AVR Micro Controllers

Felix Morgner and Manfred Morgner

April 23, 2012

This book is ongoing work. If you find errors, typos, other mistakes, please get in touch with us. We are always happy to be shown better ways, clearer views, faster solutions, lesser errors ... you name it.

If you wish to follow our ways, it would be a good idea to get a copy of ‚ATMEL 8bit AVR instruction set manual':

`http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`

In our book we will not explain the commands we are using because all explanation is already worded out in the named document. Clearly we will discuss why we are using specific commands if there are known (as in the context of the book) alternatives.

This book is mainly based on *ATmega8* because we need to push aside micro controller model specifics. At some point we have to start to deal with this problem but we try to prevent this from happening as hard as we possibly can.

We are more destined to show what can be made and done with a small micro controller. It's clear for us, that more powerful platforms are available for a similar price. But our believe is, that learning Assembler Programming leaves deeper traces if the result of ones work is amazing in face of the used hardware, like a music instrument out of a micro controller some wire and some resistors.

This book is about Assembler Programming. We do not share the view that Assembler Programming will rescue the world. Each programming language has its purpose and some of them are useful too. The reader of this book wishes to learn Assembler Programming by example and has a tendency to experiment with the real thing. We try to satisfy this kind of reader by keeping the circuits simple and so price and requirements low.

Contents

List of Figures

List of Tables

Part I

Simple Samples

Chapter 1

Light

In this chapter we will demonstrate the basics of assembly programming with AVR Micro Controllers. These samples will only require the simplest of additional hardware. If you use an Arduino, you will need only a simple wire for the first examples and the very first example nothings more than a powered Arduino.

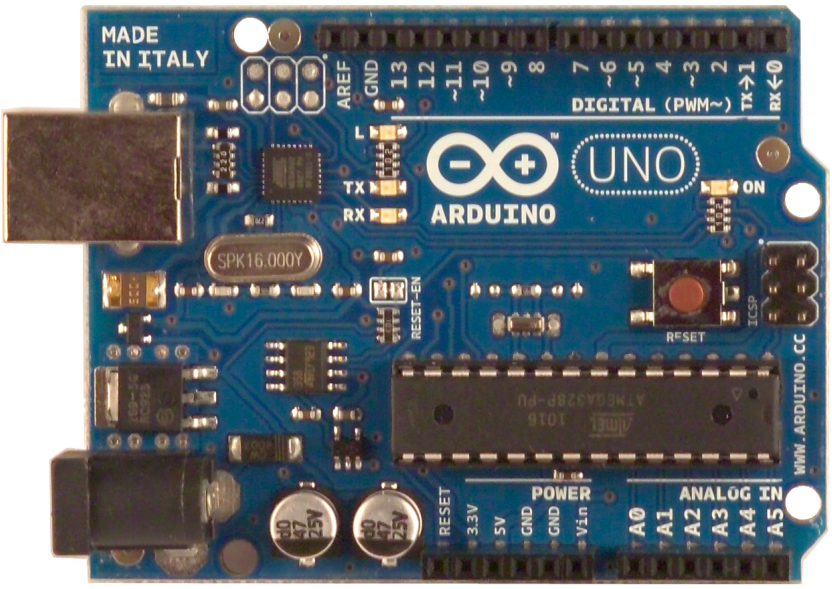


Figure 1.1: Arduino Uno

If not, you may build a circuit like in figure **??** on page **??**. The breadboard layout is shown in file `LED/S000_LED-Basic-Circuit.fz` in Fritzing file format.

As you may find out, we are focused on Free and Open Source Software. All published program

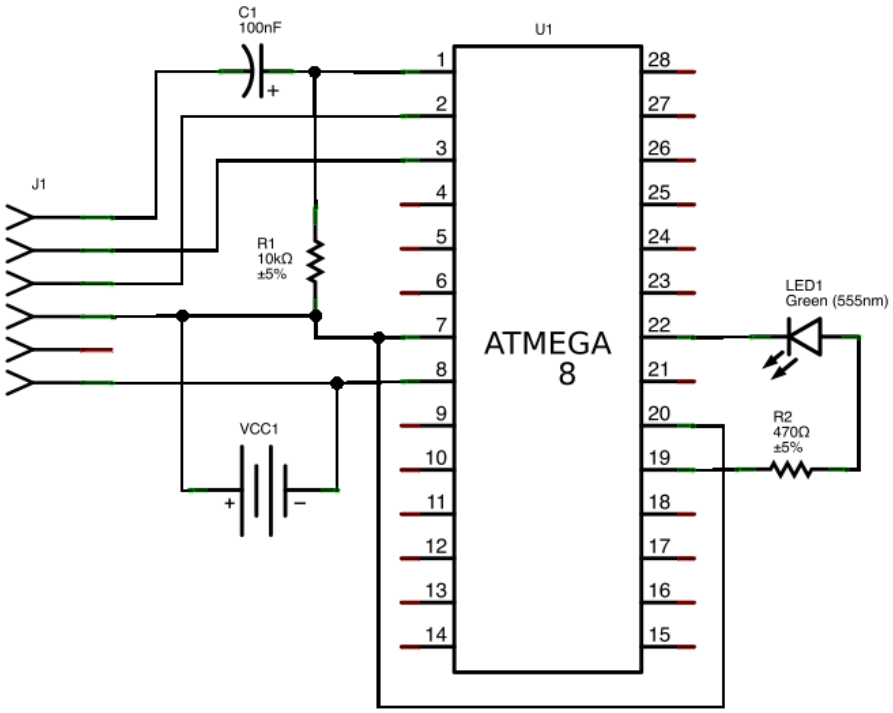


Figure 1.2: Light - Schema

code here is Free Software, the book itself and all used attachments are as free as we allowed to make them.

Also we will prevent usage of non free 'things' as hard as we can. We will not force anyone to use non free products to follow our ways. On of our main goals is an intellectual contribution to the world of Free Thinking, Libre Software and cooperative work.

1.1 Let there be light!

The first sample in this chapter is the smallest program I can think of which does something.

This program will switch on the LED on your Arduino pin 13.

Programming in Assembler means you're in control. But as you know, power requires knowledge and responsibility. If you're entering the world of assembly programming, you have absolute power if you wish to or not. Consequently you need knowledge to become responsible.

At first you need to know what 'Arduino pin 13' really means. As consequence of the Arduino design it is not pin 13 on your *ATmega8* . Knowing this is yet the half way thru. Knowing the pin is a step you need to know if are working with a plain chip. What you need to know is the micro controllers internal addressing for Arduino pin 13. To find out, look at `http://www.arduino.cc/hu/Hacking/PinMapping`.

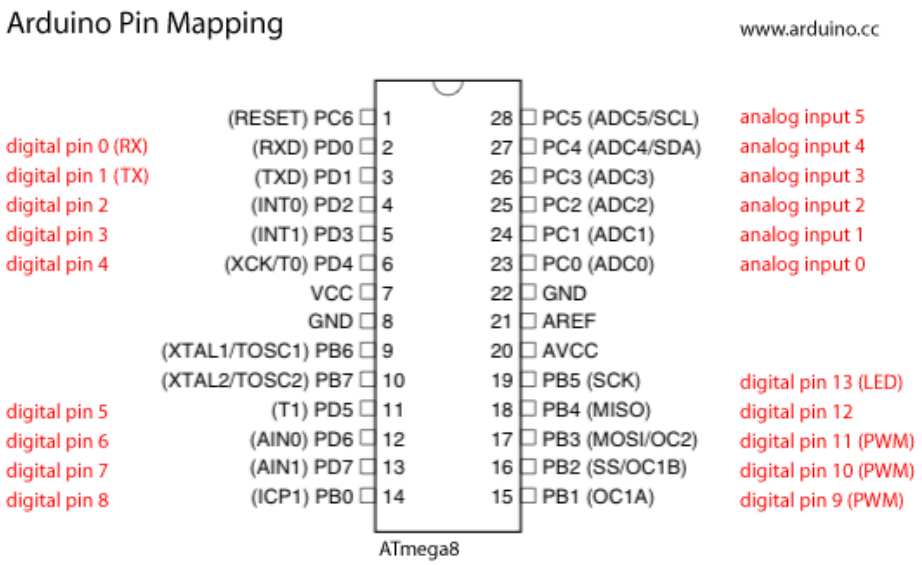


Figure 1.3: Arduino to *ATmega8* pins

After we know everything we need to know to be responsible, we generate our 8byte program that will set our 'LED 13' under power.

```
; LED/S000_let-there-be-light.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,       5
            sbi     PORTB,      5

main:
            rjmp    main
```

As simple as the program is, I believe there is some need for explanation.

At first we have to declare the type of micro controller we are intended to use with the program. This is necessary because different micro controllers do have different assignments in their inner structure and need different addressing for their components. We do this by:

```
.DEVICE atmega8
```

Next we need to declare where our world will begin. Funny thing is, we will never really know! So we are forced to use symbols to deal with this necessity. As indicated below, different micro controllers will have different inner values. But not only this, to be honest, where our program lives is due to additional effects a most uncertain thing. We come to this later in the book.

As we are forced to use symbols, we have to do so. We will set a symbol to our programs starting point later and this symbol will be named 'start' in out code. Whatever starts our program it must be informed where to goto to do so:

```
.org 0x0000
            rjmp      start
```

With '.org' (don't forget the leading dot!) we build a sequence of command positioned at the addressed position. This sequence, some times named a table, is a list of actions to do for different requests. At the moment he only request we have is to start our program and fortunately the entry for this request is expected as the first in our table.

For those who really want to know: The addressing in this table is relative to wherever it is placed in real life! So it always starts with 0. Strictly speaking, at this point we already enter the realms of a dreamworld. We don't know what really happens! But sometimes we don't need to. To declare our first magic point, we only need to postulate:

```
start:
```

'start:' is a label and represent the final address for the first memory position used afterwards. In our case the address of the first command in our program.

The next label is already behind all things we need to do in our program. It is the starting point of an unconditional infinite loop. This sequence is necessary because the processor (CPU) of our micro controller (MC) is running as long as it has power. We can't stop it, so we have to lead it into a controlled way of doing 'nothing' because we don't wish our MC to do anything after it did all things we expected it to do.

Between 'start:' und 'main:' is our program. I call this the 'first form of a standard program'. A program which runs ones after the system awakes. Such program may be of limited use, but not completely useless. And the schema of this 'first form' is the basic schema of all derived form. The program

- starts
- does something
- loops forever, possibly doing something

If the program does something inside the 'infinite loop' then this may be called the 'second form of a standard program'. A third form should be expected to pop into existence later on. But anyway. Our program of the first form is specifically designed to show some important rules for good programming.

The two commands which full fill our programs mission will do two things:

- declare pin 5 at PORTB as output pin
- set pin 5 at PORTB under power to enlighten our LED

```
            sbi       DDRB,           5
            sbi       PORTB,          5
```

Finally the never ending loop:

```
main:
            rjmp    main
```

This is all the program does and there is nothing more about it. You will discover, that this program demonstrates prudence and thrift.

A PORT of an 8bit MC controls eight bit unsung eight pins on the outside. In our *ATmega8* each one of these pins can be used to:

- Put a signal to his pin
- Read a signal from its pin which may be +5V or GND as 1 or 0
- Read a signal from its pin which may be 'not GND' oder GND as 1 or 0

Each pin as can be controlled to do one of these things freely, independent of all the other pins on the same port.

On some pins are additional features possible like

- power saving modes
- PWM modes
- analog to digital converting
- interrupts

In our program we use the command `sbi` to set pin 5 instead of the alternative command `out` with parameter `1 « 5` which would do the same but not really the same. We wish to manipulate pin 5 only! `out` would not only send 1 to pin 5 but also 0 to all other pins.

Even if we - at the moment - know that all other pins are unused, we face to situations:

1. We will become less and less sure about the usage of pins we don't use in a particular situation.
2. We don't know what happens if we send 0's to pins we don't know.

One major concept of good programming is, to do as less as any possible.

If there is nothing to be done, don't do it! Especially in micro controllers where action means energy loss! So if we need to manipulate a bit, we should not manipulate others bits except we have a good reason to do so. Currently we have not.

We don't want to wake up pins if they could sleep in peace. Because possibly, these otherwise unused pins will eat up our energy.

1.2 Get me on, Get me off

At first the enhanced schema

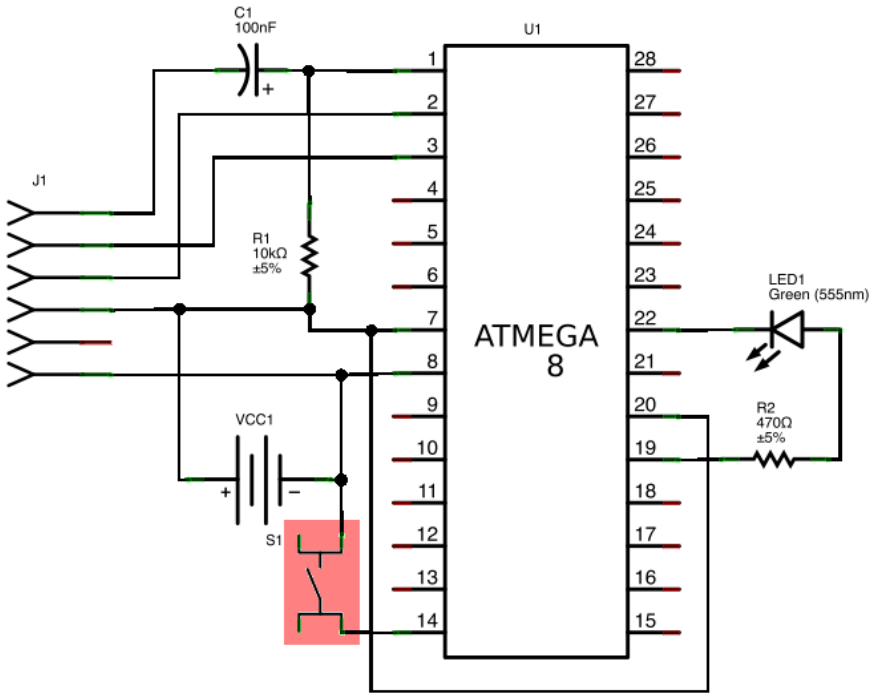


Figure 1.4: Get me on, get me off - Schema

Second the code

```
; LED/S002_get-me-on-get-me-off.asm

.DEVICE atmega8

.org 0x0000
            rjmp     start

start:
            sbi      DDRB,          5
            cbi      DDRB,          0
            sbi      PORTB,         0

main:
            sbic     PINB,          0
            rjmp     led_on
            cbi      PORTB,         5
            rjmp     led_ok
led_on:
            sbi      PORTB,         5
led_ok:
            rjmp     main
```

As you may have guessed, this code represents the 'second form of a standard program'. It does

something before the loop - ones - but also does something in the main loop. This program is really busy as we will see in the next program.

The first difference to the former program is, that we use an additional pin/bit. This time for input. To set this pin, bit 0 on port B at pin 14 of the micro controller, to input mode, we send a '0' to its data direction register (DDR). Further we send a '1' to this bit 0 of the port as if we would set the port to output +5V. But this bit is in 'output mode'. In this case, sending '1' to the bit at the port means: 'Switch on the internal pull-up resistor!'

After this, the signal read from this bit is 'ON' or '1'. To interact with the micro controller and change the signal to OFF, we need to pull-down the pin to GND.

If we do so, our program switches the light off as long as we ground bit 0, which does not make real impressive application to boast around with. For us it is great anyway because we understand what happens.

If you follow the programs flow:

1. Initialise system and devices
2. Read bit 0
3. Set bit 5 accordingly
4. Start with (2)

You may ask why we permanently output a signal that nearly never changes. Assuming our program reads the input bit one million times per second, a human will not be able to put any business to this program by flipping the switch on and off. If you pick on your signal line as fast as you can then in the eye of your program the signal nearly never changes.

Is there room for optimisation? We believe not.

Are there alternatives? Yes there are!

We could save the output status, compare the status from the input bit with the stored status the light was set to and only change the output signal if the input signal has changed.

This sounds easy, but it is not! It is not only not easy, it is dangerous, costly and complicated. And it is of no use, because you do a lot of commands without any effect.

The concept is dangerous because you may be out of synchronisation with your light. In this case you light may react inverse to your intension or does not react at all.

It is costly not only because the program would be much larger but you need to eat up one CPU register and we only have 32 of them at all.

And it is complicated because you have to synchronise two entities (the light and the status register) to generate one effect. Which is a major risk and drawback.

Because of this, we might be right in our assumption that the developers of our *ATmega8* micro controller created the chip in such a way that nothing happens if we switch on a bit that is ON already.

Another alternative may be to 'stop' the program as long as it waits for the input signal to change. This may be possible, but it lies far behind our current knowledge.

There is no command that makes the micro controller wait for an input signal. But there are ways to reach a similar effect.

1.3 Stable Decisions

For the next step we need to do something more useful. But we stay with the 'second form of a standard program': A program that does something in its infinite loop.

This program stores one of two states ;-) and sets the light on or off, keeping it according to the stored status. This explanation is slightly wrong. I try it again.

This program recognises an input signal consisting of two phases. A complete input signal consist of a LOW phase starting from a HIGH phase followed by the next HIGH phase. The signal ends with entering the HIGH phase again. We are interested only in change, as we always should.

If the Input signal changes from HIGH to LOW, our program changes the light from its current status to the opposite status. If the light was ON it will be set to OFF and vice versa. The status is only changed as reaction of changing the Input from HIGH to LOW because 'no action' at the Input device leads to HIGH status in the input bit as result of using an internal pull-up resistor who does what he is called - he pulls the input signal up to HIGH.

The electrical signal we are waiting for with our micro controller on our input bit is: Pulling the status down to LOW (GND). We may phrase:

The *Signal* we are waiting for is the *Change* form HIGH to LOW.

So for the first time we have to be aware of a dynamic process. If the change of the status is the signal, then we have to recognise if the status is change. If a status appears after the opposite status was recognised - in the past - which is then no more present (!) we have to deal with 'historical' status management and so for the first time we remember a status from one processing cycle to the next.

Finally this here is the Code to do it:

```
; LED/S004_stable-decisions.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start
```

```
start:
            sbi     DDRB,         5
            cbi     DDRB,         0
            sbi     PORTB,        0

            ldi     r16,          1

main:
            sbic    PINB,         0
            rjmp    led_keep
            tst     r16
            breq    led_ok
            clr     r16
            sbis    PINB,         5
            rjmp    led_on
            cbi     PORTB,        5
            rjmp    led_ok
led_on:
            sbi     PORTB,        5
            rjmp    led_ok
led_keep:
            ldi     r16,          1
led_ok:
            rjmp    main
```

As you can see, this Code is not easy to understand. To do better, we add symbolic names to the soup. The basics are easy:

- `.equ` means: 'a name for a value'
- `.def` means: 'a name for an entity'

So for example `DDRB` already is a number. This number is defined in an include file chosen by you device selection. In our case, whatever number is hidden behind the name DDRB it will be our Input/Output control port. So the name will be `ctl` as prefix for 'control' and `IO` short for Input&Output.

In another sample `bit` stands for 'bit number' and `Input` for 'Input' which makes `bitInput`, the bit where we read the input status.

You may define you own naming convention which better should hold throughout your project.

```
; LED/S005_stable-decisions+symbols.asm

.DEVICE atmega8

.equ ctlIO     = DDRB    ; DDRB  is our I/O control register
.equ prtIO     = PORTB   ; PORTB is our I/O output port register
.equ pinIO     = PINB    ; PINB  is our I/O input pin register

.equ bitOutput = 5       ; bit 5 is our output bit
.equ bitInput  = 0       ; bit 0 is our input bit

.equ FALSE     = 0       ; 0 will be FALSE or OFF
.equ TRUE      = 1       ; 1 will be TRUE  or ON

.def bStatus   = r16     ; the last state will be stored in 'r16'
```

As you may not have expected, this makes the soup - or code - somehow better readable and much easier to understand. Now it looks more like a higher language:

```
.org 0x0000
            rjmp    start

start:
            sbi     ctlIO,          bitOutput
            cbi     ctlIO,          bitInput
            sbi     prtIO,          bitInput

            ldi     bStatus,        HIGH

main:
            sbic    pinIO,          bitInput
            rjmp    led_keep
            tst     bStatus
            breq    led_ok
            clr     bStatus
            sbis    pinIO,          bitOutput
            rjmp    led_on
            cbi     prtIO,          bitOutput
            rjmp    led_ok
led_on:
            sbi     prtIO,          bitOutput
            rjmp    led_ok
led_keep:
            ldi     bStatus,        HIGH
led_ok:
            rjmp    main
```

Also it provides us with the possibility of changing things with reduced risk. Shortly we had to change the schema a bit to support additional ideas and we changed `bitInput` from 4 to 0. Using symbolic names this was much easier and much less risky to do because we only needed to change the definition for `bitInput`!

Even if symbols make a better reading than constants, it seems not really to be easy to follow the program flow. So at first, we should introduce a program flow chart. And for good measure two of them. We need two of them to demonstrate a major point in assembler programming.

We have to take watch about WHAT we wish to do, but equally to about HOW we are going to do it.

1.3.1 WHAT to do

1. Initialise system and devices
2. Wait for the change "input bit was HIGH and became LOW"
3. Invert LED status
4. Restart at (2)

1.3.2 HOW to do it

To get an impression on how to it, at first, we take a look at a flow diagram. This diagram shows the program flow.

As you may have found out until now, dealing with the 'human device interface' is the most complicated thing in informatics. It starts with the unbearable slowness of bio entities and does not end with humans expectations against machines. Which means the user is problem number one in software development and for this has to be the main focus! Otherwise your product will not be accepted by those who should pay our wages!

1.3.2.1 Slow users

A living entity presses a button and our device has to react. Our device may ask the input pin about 1.000.000 times per second. A human for example may do a very short pressing of a button in about 0.2 seconds. Which means, our System reads the status 'button pressed' about 200.000 times during our humans action. But what the human expects is:

Change the light ones as I press (in my view) the button ones!

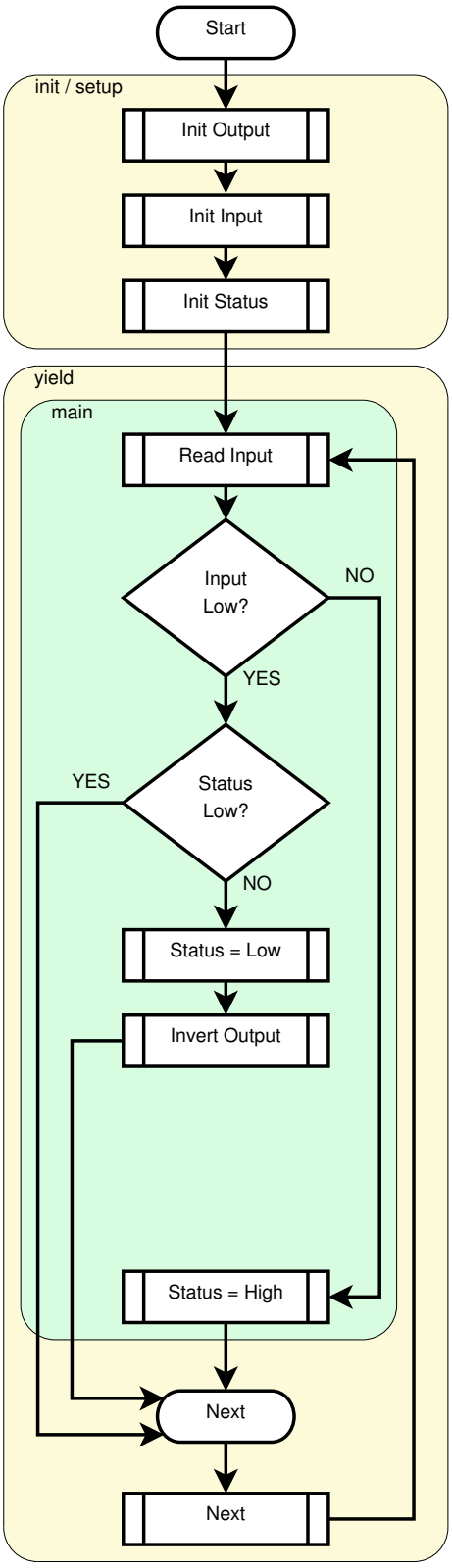
A problem we will not deal with in this phase of our development is, that electric switches may flicker during status change. Flickery leads to problems because it will generate random additional signals during the action phase. In general this is dealt with by electronically measures. Therefore we simply have to ensure that changing the LEDs status only happens ones per button press action and we will ignore any flickery event by not doing anything about it.

1.3.2.2 Fast feedback

Our feedback device is the light we control. We have no display oder acoustic device to us as feedback but the very same object that goes to be controlled. User feedback therefore simply means to manipulate the status of the light. Even if its cool in movies to do anything without any visible feedback, real living bio entities require 'immediate feedback' for every action. Otherwise they run into problems.

Figure **??** shows the signal we have to expect (idealised). You have to remember, that our micro controller reads the 'LOW' status of the signal 100.000 times or more often. We do not measure time! The natural timing unit in micro controllers is cycles. Read cycles, CPU cycles, sensor cycles. This is important to recognise especially because different clock frequencies and different voltage leads to different physical cycle durations. A program that measures a certain event by 1000 cycles will receive 2000 if the clock is doubled up.

Different clock frequencies lead to different CPU dependent cycles. Different voltage may lead to different sensor cycles, depending on the specification of the sensor in question and the characteristics of the signal to measure. Further, different voltage may lead to different absolute sensor signals, but this is another chapter.




```
start:
        sbi     ctlIO,   bitOutput

        cbi     ctlIO,   bitInput
        sbi     prtIO,   bitInput

        ldi     bStatus, HIGH

main:

        sbic    pinIO,   bitInput
        rjmp    led_keep

        tst     bStatus
        breq    led_ok

        clr     bStatus

        sbis    pinIO,   bitOutput
        rjmp    led_on
        cbi     prtIO,   bitOutput
        rjmp    led_ok
led_on:
        sbi     prtIO,   bitOutput
        rjmp    led_ok

led_keep:
        ldi     bStatus, HIGH

led_ok:

        rjmp    main
```

Figure 1.5: Stable Decisions - Flow Diagram

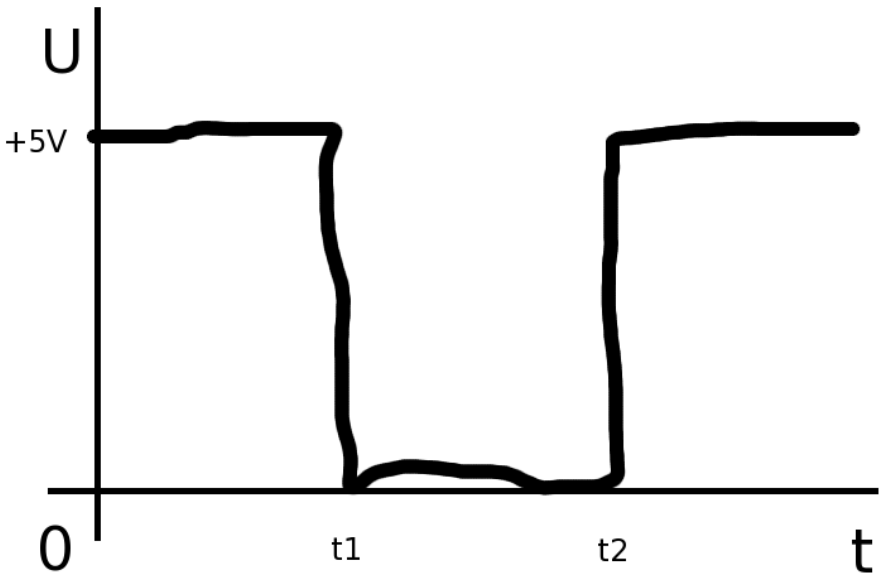


Figure 1.6: Stable Decisions - Signal Diagram

The signal we need to recognise at the moment is the switch from HIGH to LOW and back again. The LOW level will stay for an undefined time / amount of measure cycles. What we have to focus on is the change of the signal!

To deal with this there are only two moments in all this 'endless' button down phase - between 't1' and 't2' - where it is applicable to really change the LEDs status. At the beginning, as the signal changes from HIGH to LOW (t1) or at the end, where the signal changes from LOW to HIGH again (t2).

To give the human who uses our device (the user) immediate feedback about success or failure of his action, we decide to use the first phase (t1) to do all the action. So we have a simple mission: If the signal is LOW now and was HIGH the last time we looked, we change the LEDs state. This satisfies also the endless LOW state of the input signal and the change from LOW to HIGH where we have to do nothing. Not even accidentally! This way, the human feels his request immediately answered, even if he is unable to comprehend on which grade immediately it real is answered.

1.3.3 Finding the event

To find the event the signal has gone to ground, we have to be aware of the status of the signal from the previous query cycle. Which is easy but expensive. We simply use a register to store the previous input status.

We simply have to check if the last seen status was HIGH as log as the current status is LOW. The status accumulator register follows the status of the input signal after being used to query

the change event.

If the one moment we are waiting for passes, we invert the LED status. That's all

1.3.4 And some modesty

In respect of things to come, we have to be modest in our style. The 'endless loop' which keeps the micro controller running and waiting without going wild, will be used for some important things:

- It mostly contains the main program
- It possibly consist of multiple parts
- It consist of an undefined amount of functional separate program parts

So we have to ensure, to don't make any shortcut back to the 'main' label. As there can only be one Highlander, there can only be one jump back to 'main'. This jump resides at the very end of the main sequence. Like this:

```
main:

  A_begin:
            block     A or goto to end of block A
  A_end:

  B_begin:
            block     B or goto to end of block B
  B_end:

  C_begin:
            block     C or goto to end of block C
  C_end

mein_end:
            finalise
            rjmp      main
```

In our flow diagram one part has to reach the next part regardless of how the program is flowing. The rounded rectangle 'next' represents the central point behind our (currently) one part. This is the position where the next program part would follow in the same manner.

1.4 Stable Decisions Triggered

If you feel a bit uncertain about our solution or if you have the feeling that all this should be done better you may be right. We would not ensure you that the following solution is better under every circumstance, but for some application, there is a much better way.

For this way we need to introduce the concept of interrupts.

The magic behind interrupts in micro controllers is much more as in simple CPUs!

Interrupts are special mechanics in processing units to enable the execution of code at a certain time or after a certain event by interrupting the normal processing, doing something else and after this continue whatever was interrupted.

In micro processors this mechanic is much more sophisticated. In micro controllers you may interrupt the system from nothing! Meaning, it is possible to nearly shut off the whole system, interrupt it from his deep sleep, let it do something and send it back to sleep again.

Such applications are useful if energy resources are small. For example, if you wish to drive your weather station one year on a single AAA cell or less, possibly supported by solar power.

In such applications you wish to reduce power consumption of your system as far as possible. There is no need to let your micro controller do eight million cycles per second if you take a measurement every ten minutes! It may be much better so stop the whole system until the next measurement is to start.

You already may expect it. Such code does not need to loop with full processing power to do nothing, such code really does nothing while waiting:

```
; LED/S005_stable-decisions-trigger.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start
            rjmp    trigger0

start:
            ...

main:
            sleep
            rjmp    main

trigger0:
            do_it
            reti
```

This approach also has the advantage to don't need the otherwise necessary status accumulator register and consequently no comparison between former and current status. It simple needs to change the current status of the light if called.

1.5 Light Shift

Next we will start with showing off. A little bit at least. We will put three lights in a row and lighten one of them up. Each time we press the button the light will shift to the next LED. After the last light, we start with the first one.

It would be much easier to do this with 8 lights, but here we are. We will keep the existing circuit as untouched as possible and, not at last, we are constantly on he search for a challenge.

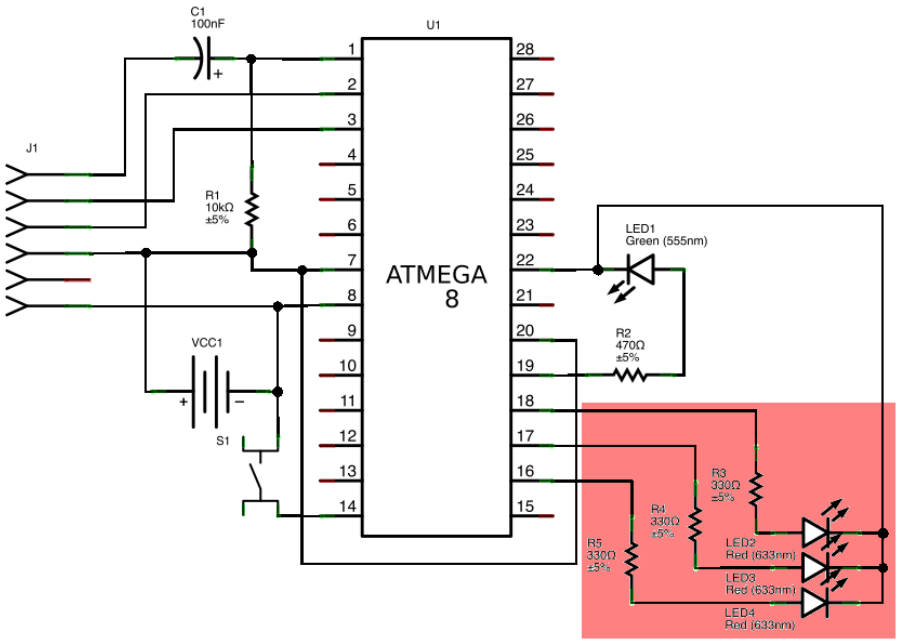


Figure 1.7: Light Shift - Schema

So here is the enhanced circuit:

As you can see, we added the three LEDs to the remaining pins on the right side of the chip, so it is easy to build in Hardware on a breadboard. For this we have to pay with a higher degree of complexity in our code

And this is the code:

```
; LED/S010_light-shift.asm

.DEVICE atmega8


.equ ctlIO          = DDRB
.equ prtIO          = PORTB
.equ pinIO          = PINB

.equ bitSignal      = 5
.equ bitInput       = 4
.equ bitLightStart  = 3

.equ mskLightShift  = 0x0E

.equ LOW            = 0
.equ HIGH           = 1

.def bStatus        = r16
.def bTemp          = r17
.def bData          = r18


.org 0x0000
            rjmp    start
```

```
start:
            ldi     bTemp,          mskLightShift | 1 << bitSignal
            out     ctlIO,          bTemp
            ldi     bTemp,          1 << bitInput | 1 << bitSignal
            out     prtIO,          bTemp

            ldi     bStatus,        HIGH

main:
            sbic    pinIO,          bitInput
            rjmp    led_keep
            tst     bStatus
            breq    led_ok
            clr     bStatus

            in      bData,          pinIO
            mov     bTemp,          bData
            andi    bData,          0xFF - mskLightShift
            ori     bData,          1 << bitInput

            andi    bTemp,          mskLightShift
            lsr     bTemp
            andi    bTemp,          mskLightShift
            brne    shift_ok
            ldi     bTemp,          1 << bitLightStart
shift_ok:
            or      bData,          bTemp
            out     prtIO,          bData

            rjmp    led_ok
led_keep:
            ldi     bStatus,        HIGH
led_ok:
            rjmp    main
```

As you can see, we have some changes to the former code. This is, in the hole, because we have to deal with four instead of one lights. One light (the one we toggled last time) will serve as signal light. It is lighting up after our micro controller has finished booting. The other three LEDs will be used to shift the light. In our case this means we start off with one of the three lights ON and each time we press our button the next light lights up whilst the former light shuts off.

Our code reflects the higher complexity at the first look with more constants and more named registers. Already we are using 10% of all available registers and 20% of all registers usable for generic purpose. This is bad news! Hopefully our resource consumption will not grow with constant speed.

Names for constants (not resources) we need besides the ones from the former sample are

```
.equ bitLightStart = 3     ; as in  0b00001000

.equ mskLightShift = 0x0E  ; builds 0b00001110
```

`bitLightStart` is the number of the pin on the output port where the LED light of our light chain resides. A bit will be shifted this time to the left. The other LEDs are expected to use the two lower pins (2 and 1). Pin B0 on the *ATmega8* can be found on the other side of the chips, so for our physical breadboard circuit it's a bit too far away to use it.

Please remember, such decisions are no fun. In reality you may sometimes be driven to compensate in software for simplification of hardware as PCB layout, different chip pinouts and so on. So we accept this situation as example of what may happen in real life.

`mskLightShift` is a bit mask containing bits at all pins of the output port where our light chain is connected to. We need such a bit mask because we need to shift the 'LED ON' bit around but won't shift any other bits found in the port data. So we use this bit mask to

- mask out the relevant light chain bits from the rest
- mask out a too far shifted 'LED ON' bit (logical overflow)
- mask out the light chain bits in input data after querying our PORT

Named registers (resources) we need in addition to the former program are:

```
.def bTemp          = r17
.def bData          = r18
```

`bTemp` will be used to shift our 'LED ON' bit around but `bData` has to give us the 'big picture' about our ports status. First before, second after shifting the 'LED ON' bit.

We start, as usual, by initialising the micro controller where it needs to be initialised. This time, we need to set all pins on our port the same time because the alternative is (for the moment) much too expensive. So we build a bit mask to send it to our IO ports data direction register (DDR):

`mskLightShift` combined with `1 « bitSignal` makes `0b00101110`. Sending this byte to DDRx will set pins 1, 2, 3, 5 to output and all other pins to input mode.

```
start:
            ldi     bTemp,          mskLightShift | 1 << bitSignal
            out     ctlIO,          bTemp
            ldi     bTemp,          1 << bitInput | 1 << bitSignal
            out     prtIO,          bTemp
```

Then we need to do three things by sending a byte to PORTx:

- Switch ON LED on pin 5 (`bitSignal`)
- Set pin 4, our input pin, to 'pulled up' (`bitInput`)
- Set all other pins to off/offline

All this is done by mixing all active bits together and sending the mix `(0b00110000)` to the port. After doing so the signal LED is on, the light chain is off. Until `clr bStatus` nothings more has changed. Also the rest with and after label `led_keep` is kept the same.

The changed part is this:

```
            in      bData,      pinIO
            mov     bTemp,      bData
            andi    bData,      0xFF - mskLightShift
            ori     bData,      1 << bitInput

            andi    bTemp,      mskLightShift
            lsr     bTemp
            andi    bTemp,      mskLightShift
            brne    shift_ok
            ldi     bTemp,      1 << bitLightStart
shift_ok:
            or      bData,      bTemp
            out     prtIO,      bData
```

All magic resides in these lines. It's not too much, so we simply explain it sequentially.

We need to get the ports state and unfortunately we have to understand more about this than is good for the program. We have to send back the whole data byte, modified only in the part where the LED chain state changed. BUT (!) we also need to ensure, our input pin keeps his pull up resistor active. Otherwise we loos our connection to the outer world!

Next we need to split the read byte/data in two parts. The one containing constant data and the other one containing the bits to be shifted.

Phase one therefore copies `bData` to `bTemp`, then masks out all bits dealt with in the bit shifting process and finally adding/regenerating the 'pull up' bit for port `bitInput`.

Phase two masks out all bits not related to the bis shifting operation, shifts the remaining bit and masks the result again with the same mask. If this operation leads to a result of ZERO/EQ/NULL, then we have shifted out our bit and need to insert a new one at the start position inside the byte where the bit starts its decent, at: `bitLightStart`.

Phase three then combines our static bits with our manipulated one and output all together to our combined input/output port.

Chapter 2

Time

2.1 Instable Elements

The longest time we tried to avoid the simplest demo in most Arduino beginners sets. The blinking light demo! Some of our readers may have ask themselves where the problem should be. Now is the moment to explain.

We do not want to make a fuss with code we would have to be ashamed of but couldn't bring us to start with timers and interrupts before introducing the most basic principles of programming. Please remember, using assembler language brings us in a position of power which forces us into reliability.

There is no way a reliable programmer would use 4.000.000 NOPs ('no operation' operations) twice to let a light blink ones per second. Also, we hope, no one reading until this point would keep it as responsible to enter a loop, busying our poor micro controller to wait half a second by wasting four million CPU cycles.

So we had to experiment enough to enter the reign of timers and interrupts. Staying our ground not demonstrating bad code, keeping the book pure, we don't show only a part of a single bad example in code. You may remember it in your dreams!

To cut a long story short. Here its the code:

```
; LED/S020_instable-elements.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            ...
main:
            rjmp    main
```

Part II

External devices

Chapter 3

Shift Registers

3.1 SRAM to Shift Register

Sending SRAM content/data to shift registers has some applications. Some of them are

- Light chains
- Raster displays