Programming AVR-AtTiny controllers under Arduino

Part 1 – Methodology and basic tasks

Traditionally programming an AVR microcontroller was accomplished by using Atmels AVR-Studio and a compatible ISP programmer.

Now a large community have gained some experience by playing with Arduino and they are doing applications using shields. It is easy now to do PCBs, special 3D packaging, hence the need to be able program a small microcontroller that can do a dedicated job is increasing, and it can be a lot of fun too.

The objective of this document is to show to users of Diduino boards (or any other Arduino compatible boards) with some experience how to program the AVR "Tiny" microcontrollers in 8 and 14-pin packages and move from "Arduino control" to "C Tiny control".

This document focusses on AVRs 8-pin and 14-pin microcontrollers (i.e. AtTiny13[1], AtTiny45, AtTiny85 – AtTiny44, AtTiny84 etc), similar to the Microchip PIC available for over 15 years.

An Arduino compatible board can be used to program these Tiny-8pins and Tiny-14pins. The AtTiny microcontrollers are notably cheaper than a full featured ATmega328 used in the Arduino or Diduino but they also come with significantly less flash and RAM memory.

While you can in fact just download a lot of Arduino sketches directly to an AtTiny microcontroller it is mostly impractical due to the memory size restrictions the AtTiny imposes. With 2k – 8k flash memory, often the very simplest sketches become too large to load onto the AtTiny of your choice. It is not only a question of memory space. Embedded processors have real time and power constraints.

You cannot really play the "Tiny" game if you insist to program using the Arduino functions (digitalWrite, etc.). Several Arduino documents found on the Web show to get rid of these memory ineffective functions. In this document we try to give a more complete and thorough understanding of the process of migrating or creating an application on the AtTiny microcontrlollers. The examples used might look confusing and complicated at first – but they are not. Please make sure you understood the examples before proceeding in the document.

The EPFL MOOC "Comprendre les Microcontrôleurs" and the associated documents (www.didel.com/coursera/LC.pdf in French) is a good base for understanding both worlds and being capable to write efficient "Tiny" programs. But we do not assume this and start here from scratch.

C++ experts will see it is a different world.

Ways of physical programming

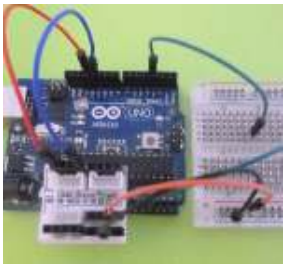
There exists a multitude of ways to interface an AtTiny microcontroller for programming. While hobbyists and hackers use an Arduino board with a breadboard and a handful of jumper wires, the pros often have more permanent and robust solutions with ZIF sockets. We will use the Didel AtTiny programmer, available from Boxtec 2 which is designed as semi-professional tool for programming many different AtTiny chips.

---

[1] Note AVR documentation writes ATtiny, we write AtTiny in all our documents

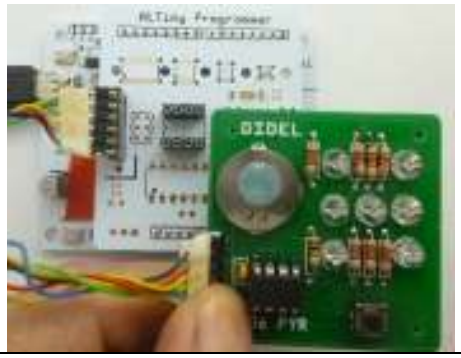
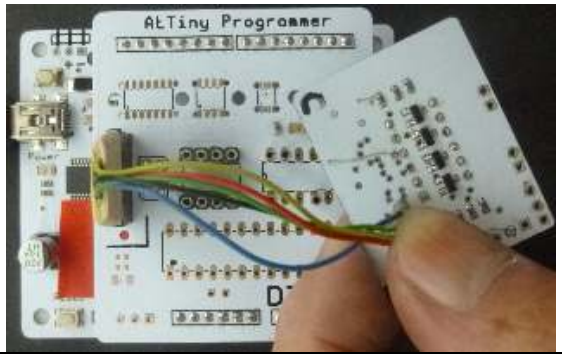
2 See www.didel.com/diduino/AtTinyProgrammer.pdf
or in French www.didel.com/diduino/ProgrammerUnAtTiny.pdf

A typical project is shown below. An autonomous sensor or clock uses a Tiny-8 programmed on the Arduino that has been used to develop the program. In a first version the program could have used Arduino functions, but all these functions have been removed in favour of a "pure C" program that runs on the AVR AtMega328 of the Arduino/Diduino. Changing few definitions makes the program compatible with the selected Tiny and programming will be successful.

		
Develop and test	Program a Tiny-8 or Tiny-24	Run your board

In system programming (ISP) 3 avoids moving the microcontroller between the programmer and the board. Usually, many development cycles are required: write code - compile – program to chip – test and find out code doesn't work so your return to first step.

As example, the DePyr kit soldered by pupils (picture) can be reprogrammed with simpler C programs they understand. No connector,a cable connect the dice during programming. Connector with a 1.27mm pitch is used for small SMD boards.

An important constraint with ISP programming is that the signals of the application do not disturb the programmer that sends the CK and MOSI signals and reads the MISO from the AtTiny. Evaluate the sink and source current on these pins when designing the schematic!

The AutoISP adapter of Matthias Neerarcher 4 is a good way to be indenpendent of these wiring constraints when using DIL packages.

Developing any software consists of many learning steps and tests. This document cannot cover all aspects of this complex process. A fair understanding of C is required and you should have written non-trivial Arduino programs. You need also a good understanding of logic operations and you should be able to decode simple logic diagram and be willing to replace "off-the-shelf" libraries by sets of instructions that access directly the internal registers.

We recommend to do simple programs to test all features separately. Debugging of the end system is often difficult, if even possible at all. See our tricks at the end of this document.

AVR AtTiny-8 (13A, 25, 45, 85) and AtTiny-14 (24, 44, 84)

These processor have different memory sizes. We do not care, we will show how to do a lot in 1k bytes and 64 8-bit variables (the AtTiny13A), but you have to get rid of most if not all your

3 *http://en.wikipedia.org/wiki/In-System_Programming*
4 http://forum.boxtec.ch/index.php/topic,2779.0.html

Arduino habits. You know the layout of the Arduino, with 3 PORTS B, D, C whose pins are labeled 0 to 19.

You will find also Arduino pin numbers for the Tiny-8 and 14. Forget about them.

We will work only with PORT bits. The layout of the Tiny-8 and Tiny-14 is the following:

```
    Tiny-8
                +-\/-+
  Reset     PB5  1|    |8  Vcc
   ADC2     PB3  2|    |7  PB2 SCk  INT0  ADC1
(test led)  PB4  3|    |6  PB1 MISO Pwm
            GND  4|    |5  PB0 MOSI Pwm
                 +----+

No connection to reset
```

```
  Tiny-14        +-\/-+
                VCC  1|    |14  GND
                PB0  2|    |13  PA0  ADC0 (test led)
                PB1  3|    |12  PA1  ADC1
       Reset    PB3  4|    |11  PA2  ADC2
    INT0        PB2  5|    |10  PA3  ADC3
    ICP1        PA7  6|    |9   PA4  SCK   T1
PWM OC1A MOSI   PA6  7|    |8   PA5  MISO OC1B PWM
                          +----+
```

Deciding which pins to use as control signals for your application is not evident. Every pin, in addition to be a programmable input/output, has possible internal connections, sometimes to the A/D converter, to a timer function, to an interrupt line, etc. Make sure to check data sheet if you have special constraints or requirements. On the diagrams above, we marked the SPI signals used by the ISP programmer, the 2 A/D channels, the simple interrupt input and 2 PWM channels.

Note: we will not use the Reset pin. It implies modifying the fuses and it is then difficult to program again. See ScratchMonkey 5 for solutions.

Pins and Ports

Let us start with how to blink a led connected to PB4 pin3 on the Tiny-8 or to PA0 pin 13 on the Tiny-14, not using the Arduino facilities. If you do not understand the #define statement used, make sure you read up on it before proceeding 6 .

Tiny-8

```
#define  bLed   4 // Bit 4 on PORTB
#define  LedOn   bitSet   (PORTB,bLed)
#define  LedOff  bitClear (PORTB,bLed)
#define  LedToggle  PORTB ^= (1<<bLed)
```

Tiny-14

```
#define  bLed  0  // Bit 0 on PORTA
#define  LedOn   bitSet (PORTA,bLed)
#define  LedOff  bitSet (PORTA,bLed)
#define  LedToggle  PORTA ^= (1<<bLed)
```

Pins direction is usually defined in the Arduino setup() function, but as we try to avoid the loading of the heavy Arduino environment we will not use it here. Real C programs have only one main() loop.

```
int main () {
  DDRB = 1<<bLed
```

```
int main () {
  DDRA = 1<<bLed
```

The Arduino loop() function is replaced by a `while(1)` or a `for(;;)` statement, resulting in the same endless repeating of the enclosed code.

```
  while(1) {
     LedToggle ;
     DelMs (500);
  }
}
```

```
  while(1) {
     LedToggle ;
     DelMs (500);
  }
}
```

The function DelMs() replaces the Arduino delay() function. The Arduino delay() function uses a timer, interrupts, 200 bytes of code and is anyway a blocking function. We will see later how to do the delays with a timer.

5 http://microtherion.github.io/ScratchMonkey
6 http://en.wikipedia.org/wiki/C_preprocessor

Notations

As usual in C, constants are upper case, at least first letter for readability reason.

Fuctions start with an upper case, and are a verb that express the action.

Variables are lower case. Upper cases are preferred here to underline for subnames junction. There exist other valid conventions for writing function or variable names in C, this is just the one we used throughout this document.

A bit name in a PORT or a flag starts with a "b" This makes clear it is not a package pin or an Arduino pin. A led connected to Tiny-8 pin 3 is bit 4 of PORTB, Use a #define bLed 4

Initial #defines must give names to the register bits. The associated pin number may depend on the package. --- **Do not confuse bit numbers, Tiny pins and Arduino pin numbers!** ---

Delays

Repeating a no operation instruction (NOP) in a loop is the way to loose time. The processor runs at e.g. 10 Mhz, simple instructions takes around 0.1 us. The compiler generates many instructions for a for(){}, a while(){} block.You can study this with an oscilloscope. Let us use a chronometer to measure 1 second and calibrate the 1ms delay used as a reference.

Note that we test the programs on an Arduino/Diduino board running at 16 MHz. The Tiny may run later at 20, 10, 9.56, 8, 1 MHz. You will have to correct the count.

```
// Blink.ino   1Hz blink on Arduino, in pure C
#include <avr/io.h>  // know names as PORTB DDRB ..
#define Calib1ms 900  // 1ms at 16 MHz
void DelMs (unsigned int dm) {  // max count 65535
   for (volatile int i=0; i<dm; i++) {
      for (volatile int j=0; j<Calib1ms; j++) {}
   }  // 900 loops for 1 second
}
#define bLed 5  // portB pin 13
#define LedToggle PORTB ^= (1<<bLed)

int main () {
  DDRB |= (1<<bLed);  //PB4 out
  while (1){
     LedToggle ;
     DelMs (500);  // 0.5s half period
  }
}
```

The code size is 248 bytes, below the 466 of the BareMinimum Arduino program. Most of this code is for initializing the vectors. The BareMinimum here is still 176 bytes.

We can use the same program for any AVR processor, defining which pin on which register must blink, and adapting the Calib1ms value to the clock frequency. On a 10 MHz Tiny–8 we prepare.

```
// Blink.ino   Tiny-8  13/25/45/85  blink PB4 pin 3
#define Calib1ms 560  // 1ms at 10 MHz
#include <avr/io.h>
#define Calib1ms 560  // 1ms at 10 MHz
#define bLed 4   // PB4
#define LedToggle PORTB ^= (1<<bLed)
void DelMs (unsigned int dm) { // dm>0 delay in ms
   for (volatile unsigned int i=0; i<dm; i++) {
      for (volatile unsigned int j=0; j<Calib1ms; j++) {}
   }
}
int main () {
  DDRB |= (1<<bLed);  //PB4 out
  while (1){
     LedToggle ;
```

```
      DelMs (500);
    }
  }
```

Using the Arduino as ISP facility and the Didel AtTinyProgrammer, as documented on http://www.didel.com/diduino/AtTinyProg.pdf, the Tiny-8 you have selected is blinking on the AtTinyProgrammer led.

For the Tiny-24, the pin connected to the AtTinyProgrammer led is A0.

```
#define bLed 0    // PA0
#define LedToggle PORTA ^= (1<<bLed)
```

Of course, modify the setup:

```
DDRA |= (1<<bLed);  //PA0 out
```

Note: If you want to efficiently develop and test AtTiny programs, you should have two Arduino/Diduino boards. One is in the mode ***AVR ISP*** and ***ATmega328***, the other carries the programmer and the AtTiny (or the cable toward your board), and is in the mode ***Arduino as ISP*** and ***AtTiny25- 1MHz*** (or other). You cut, paste and edit programs for going from Arduino to AtTiny.

Do not forget to save all versions!

It is convenient to test your Tiny program first on the Arduino itself. The difference will be the change of a few #defines, to take care of the pins assignement and processor speed.

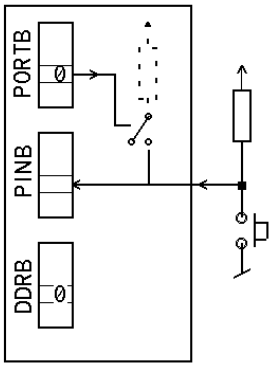
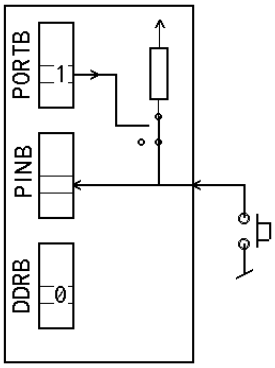
Change 2 lines on above program to test on your Arduino 328 at 16 Mhz, in order to blink the usual pin 13. Of course, you must have selected 328 board and AVR ISP programmer.	`#define Calib1ms 900  //  16 MHz` `#define bLed 5     // PB5 pin 15`

Get a key

Buttons are wired to an input with a pull-up (active low) or a pull-down (signal active-high). When the switch is open, the other voltage must be applied to the microcontroller input. This should appear in the definitions:

```
#define bButton 2  // PB2  Tiny-8
#define ButtonOn (!(PINB & (1<<bButton)))  // active low
```

Once defined, the program must refer only to the functionnality: ButtonOn, ButtonOff.

Pull-up is preferred for several rasons. One is that a pull-up mode is available on the AVRs: just set the output register bit to `PORTB |= (1<<bLed)` When there is no pull-up, touching with the hand will change the level (not dangerous in our humid countries). If it influences the software, the reason is clear.	External pull-up 2-100k 	Internal Pull-up ~50k 

Just copying a button to a led does not need a delay or debounce. Program on Arduino is:

```
// Button.ino   1Hz blink on Arduino, in pure C
#include <avr/io.h>    // know names as PORTB DDRB
#define bLed 5  // portB pin 13
#define LedOn  bitSet (PORTB,bLed) //active high
#define LedOff bitClear (PORTB,bLed)
```

```
#define bButton 2  // PB2  Tiny-8 or AtMega328
#define ButtonOn (!(PINB & (1<<bButton)))  // active low

int main () {
  DDRB |= (1<<bLed); //PB4 out PB2 In
  while (1){
     if (ButtonOn) { LedOn;  )
     else          { LedOff; }
  }
}
```

If you are confused with these notations, you should not continue before all this makes sense to you. Arduino hides the use of logic operations, #defines, pull-ups etc. from you which is a great help for beginners but comes at the price of heavy memory consumption! Using a microcontroller for small real time applications implies to program with instructions close to the micro as we do here.

Push-buttons have 1-5 ms bounces. One way to avoid bounces is to read them every 5ms. .

As a test program, let us program an on/off push button: press--> on, press again --> off. It is clear that bounces are not acceptable.

The program waits for the button to be pressed and released.

Pins are inputs at start-up; they do not need to be declared in the set-up. If you do it anyway it will help to make your code easier to read and understand, but it is strictly not necessary.

```
// BouttonTog.ino
.. definitions as above and delMs() function
int main () {
  DDRB  |= (1<<bLed);  //PB4 out PB2 in
  PORTB |= (1<<bLed);  //pull-up on PB4
  while (1){
     while (!ButtonOn) {delMs (5);)
     while ( ButtonOn) {delMs (5);)
     LedToggle;
}
```

This is a blocking program. As soon you use a for or a while loop, execution time may be long.

The if()statement is not blocking, you go through, just testing the condition.

Non-blocking tasks

Non-blocking means that the execution will go through a small number of instructions, and continues with the memory of what happened. The next passage checks for the changes. For a button, one compare the previous state and the new one. This is the way to detect an edge.

If the scanning is too frequent, bounces will be seen. If it happens not enough frequent, a short depress action may not be seen and go unnoticed by your application. A flag is activated at the moment the key is depressed. This flag is cleared by the task waiting for the key.

```
#define bButton 2  // PB2
#define ButtonOn (!(PINB & (1<<bButton)))
byte  buttonFlag=0;
byte  prevPous1On = 0 ; // active low,  seen as a boolean
if (ButtonOn && (!prevButtonOn)) {buttonFlag =1; }
   prevButtonOn = Button;
```

Note that `(ButtonOn && (!prevButtonOn))` is true when the button is just depressed. 5ms later, it is still depressed, but the copy is not any more at inactive state.

As a program example, we decide to change between two different blinking periods when depressing the button. For this we need a non-blocking blinking function. If the main loop is 5ms, blinking at 1 Hz implies to wait 100 cycles (0.5s) before toggling the LED.

We need 2 period values, a variable will do the selection.

```
byte cntPeriod;  // count by 100 for 1 Hz
byte period=100;     // period, unit is 5ms
byte periodSelect;  // 0 or 0xFF indeed a boolean:  0 or not 0
The instructions executed every 5ms are
if (cntPeriod>period) {
    cntPeriod=0;
    LedToggle;
}
```

The main program loop of 5ms tests the button and, if depressed, sets the buttonFlag variable. The periodSelect bit is changed if buttonFlag is set. The required period value is updated according to the periodSelect value. The program loop is as follows:

```
  while (1) {
     delMs (5);
// test button
     if (ButtonOn && (!prevButtonOn)) {buttonFlag =1; }
     prevButtonOn = Button;
// blink
     if (cntPeriod>period) {
        cntPeriod=0;
        LedToggle;
     }
// decide what to do if button request
     if (buttonFlag) {
         buttonFlag =0;
         periodSelect ^= 0xFF;  // complement
         if  (periodSelect) {   // if true
             period = 100;  // fast
         }
         else  {period = 600;}
     }   // end buttonFlag
  }      // end while
```

Add the definitions and delMs() function and test it on your Arduino. Change the definitions and download the program to the Tiny. If you already tested the LED and button, it is sure it will work..

You may find this program complex. Notice how well structured it is, and it does prepare us to put these instructions in an interrupt loop.

This program is named ButtonBlinks.ino and uses 322 bytes. It is available, as all the programs of this document under www.didel.com/AtTinyPorgamming.zip

Short and long actions on a button

One cannot connect many buttons on a small Tiny. Let us mention several tricks.

Recognizing short and long pushes just needs one counter and one flag.

The counter can be 8-bits, measuring durations up to 5x255 = 1275ms. The counter must saturate at 255 and just checking if the count is at its maximum allows us to distinguish short and long (>1.2s) push actions. We can also activate a flag when the count is over 0.5s (a short depress is 0.1s)

It is however not so easy, because there is a state when the button is depressed and time is counted, and a state that follows when the switch is released and the action related to the duration must be done. A similar problem that requires a state machine is seen later.

How many times depressed?

Recognizing a number of pushes can be specially useful at power-up. Depressing 1,2,3 times will change parameters or start different programs. A counter will count the short actions, a long period of time (1.2 seconds) after an action will set-up a flag. A state machine is the simplest

way to go through the states "waiting for a key", "waiting for key released", "waiting for a possible new key" and "setting the flag meaning a new count is available". Details in www.didel.com/diduino/DemosMultiples.pdf (in french).

Is a 5ms loop too slow for my sensor?

The loop can run much faster or slower if requied. In a 100 microsecond loop, you can execute more than 100 lines of code (800 AVR instructions at 16MHz). You just need to adapt the counts, as in the blinking example, to have the adequate rate for every task.. Be aware that replacing a byte counter by an int counter will add ~20 bytes to your code.

PWM and PFM

PWM (Pulse Width Modulation) is well known. It is easy to program as a blocking task. The AtTinys internal timers allows to program 2 PWM channels on defined pins. There are several registers to initialize. A blocking PWM function is easy to write. PWM is not the solution for the geared motors you may wish to use in your application. These motors need 30% PWM to start.

PFM (Pulse Frequency Modulation) must be used with 2-5 ms full pulses that counters the frictions, makes the motor start and do a fraction of turn. The motor will spin as slowly as required, of course with a cogging partly absorbed by the gearbox.

PFM is one simple task that can be done in the 5ms scheduler loop we have seen before.

Note that 255 motor speed values is a non sense. 10 or 20 are enough, Maybe if you need you 1/50, 1/200 of max speed, but for what? For adjusting the position of a robot or mechanism according to a sensor direct pulses may be better.

Let us show how to program a PFM of any resolution.

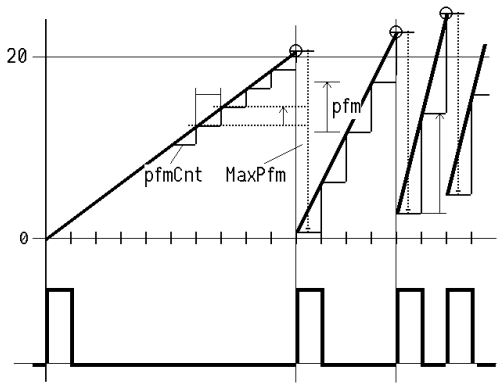
We need to define for one motor (easy to add as many as you want on any pin):

```
#define bForw 2  // PB2 Tiny-8 or Arduino PortB
#define bBack 3
#define Forward   (bitSet   (PORTB,bForw); bitClear  (PORTB,bBack))
#define Backward  (bitClear (PORTB,bForw); bitSet    (PORTB,bBack))
#define Stop      (bitClear (PORTB,bForw); bitClear  (PORTB,bBack))
```

The PFM task , forward only, need 2 variables.

The algorithm add the pfm value to the pfmCount. When the pfmCount overflow, the output is set for one cycle and M axPfm is subtracted.

For 2 motors, 2 directions, add three similar set of instructions and switch to the good one.




```
#define MaxPfm  20
byte  pfm;  // 0 .. MaxPfm
byte  pfmCnt = 0;
```

The task to do every 5ms (min 2ms for a small motor) is

```
pfmCnt += pfm ;
if (pfmCnt > MaxPfm) {
   pfmCnt -= MaxPfm ;
```

```
      Forward ;
   } else { Stop; }
```

Easy to finish and test this TestPfm.ino program.

The test can be done on a led pin. You will see the blinking at low PFM, but the cogging on a gear motor will not be perceptible.

You can find more details on www.didel.com/diduino/PwmPfm.pdf (in French).

EEPROM

Saving data on EEPROM and get it back at power-up is frequently required with AtTiny applications. Write takes few ms, read is not blocking.

One need 2 functions: WriteEE, ReadEE we take from AVR doc.

```
void WriteEE (byte ad,byte da) {
  while (EECR & (1<<EEPE)) {}
  EEADR = ad; EEDATA = da;
  EECR |=(1<<EEMPE); EECR |=(1<<EEPE);
}
```

```
byte ReadEE (byte ad) {
  byte da;
  EEADR = ad;
  EECR |= (1<<EERE);
  return da;
}
```

Our test program converts the time for depressing into a blink period and saves the value on the internal EEPROM. The value is then restored at reset or power-up. For the first execution, the EEPROM position may contain a zero. Test and change it, or make sure blink does not block for a zero period.

Measuring the depress time and storing the value needs a state machine.

```
// TestEEPROM.ino   Blink at different speed and  store the period
#include <avr/io.h>
#define Calib1ms 900  // 1ms at 16 MHz
void delMs (int dm) {
   for (volatile int i=0; i<dm; i++) {
      for (volatile int j=0; j<Calib1ms; j++) {}
   }
}
void WriteEE (byte ad,byte da) {
  while (EECR & (1<<EEPE)) ;
  EEAR = ad; EEDR = da;
  EECR |=(1<<EEMPE); EECR |=(1<<EEPE);
}
byte ReadEE (byte ad) {
  EEAR = ad;
  EECR |= (1<<EERE);
  return EEDR;
}
#define bLed 5 // pin 13 bit 5 PORTB
#define LedToggle PORTB ^= 1<<bLed
#define bButton 2  // PB2 / Ard pin 10 // active low
#define ButtonOn (!(PINB & (1<<bButton)))

byte period, cntPeriod;
byte cntButtonDuration;
byte buttonState=0;
#define AdEE 0  // adress of saved period in EEPROM

int main ()  {
  DDRB |= (1<<bLed);  //led out
  period = ReadEE(AdEE);
  if (period==0) period=20; // first time after progr
  while (1){
     delMs (5);
         // blink
     cntPeriod++;
```

```
      if (cntPeriod >= period) {
         cntPeriod=0;
         LedToggle;
      }

          // handle button depress time
    switch (buttonState) {
     case 0:  // wait for button
       if (ButtonOn)  { buttonState = 1; }
       break;
     case 1:  // measure depress time
        if (!ButtonOn)  { buttonState = 2; }
        else {
          if ( cntButtonDuration<255) {
            cntButtonDuration++;
          }
        }
        break;
     case 2:  // update period and save
        period = cntButtonDuration;
        WriteEE (AdEE,period);
        buttonState = 0;
        break;
      }  // end case
    }    // end while
  }      // end main
```

Analog inputs

The `analogRead();` function, easy to use, always available, is one of the best Arduino invention. But it takes 80 bytes for a very simple work.

Let us do it in 20 bytes using the 8-bit option. Why not the usual 10 bits? In order to make a good use of the 2 low bits of the converter, you need a very clean card layout and shielded connecting cables.

For simple applications, there are 2 or 3 control register to handle, and the 8-bit result is read in `ADCH` register. Configure – start conversion – wait – read result. It is alway the same scheme,

Enable the ADC and set one of the 7 clock frequency for the converter (0 divide by 2 for low main clock, 7 divide by 128 for 20 MHz AVR). It does not seem to be a critical value.

In the set-up

```
ADMUX  = 0x60 + 0 ;  //  8-bit mode,  channel A0 (RC0 pin 14)
ADCSRA = (1<<ADEN) + 5;  // 16MHz, try divide by 32
```

In the loop

```
ADCSRA |= (1<<ADSC) ;   // start conversion
while  (ADCSRA & (1<<ADSC)) ;  // wait till finished
byteValue = ADCH ;
```

That's all !

Do not be surprised with the 0x60 value. The AtMega328, the Tiny-8 and the Tiny-24 have differences in bit positions for the 2 or 3 concerned registers. We prefer to use masks out of AVR data sheets and not e.g. the LDLAR name that has not the same value for the different models.

The program for the Tiny-8 has not the same initialization as for the Arduino 328.

Let us give as an example a minimalist test that lit a LED above a given level.

```
//AnaRead on Tiny-8  220 bytes
//Led on pin PB4 is on if byteValue is e.g.  > 100

#define bLed 3  // led on PB4 pin 3
#define LedOn  bitSet (PORTB,bLed) //active high
#define LedOff bitClear (PORTB,bLed)

byte byteValue;
int main() {
  DDRB = (1<<bLed) ;
  ADMUX  = 0x10 + 0 ;  //Tiny-8 8-bit mode, + channel 0 1 2 3 for RB5 RB2 RB4 RB3
  ADCSRA = (1<<ADEN) + 4;   //ADEN + prescale 16
  while (1) {
    ADCSRA |= (1<<ADSC) ;   // start conversion
    while (ADCSRA & (1<<ADSC)) ;  // wait
    // result is now available in ADCH
    if (ADCH>100) { LedOn; }
    else          { LedOff;}
  }
}
```

The program for the Tiny-14 has also a different initialization.

```
//AnaRead on Tiny-14  224 bytes
.... se above
int main() {
  DDRB = (1<<bLed) ;
  ADMUX  =  0 ;  //Tiny-14  channel 0 ..7  for RA0 .. RA7
  ADCSRA = (1<<ADEN) + 5;   //ADEN + prescale 32
  ADCSRB = 0x10 ;  // Tiny-14 8-bit mode
  while (1) {
  ... same as above
```

End of Part1

Part 2 plan to cover the following topics:

Watchdog

Sleep

External interrupt

Encoders

Timer2 and its use as a scheduler

SR04/05 ultasonic sensor

Serial communications

Debugging helps.

jdn 150222