AVR-AtTiny programming under Arduino

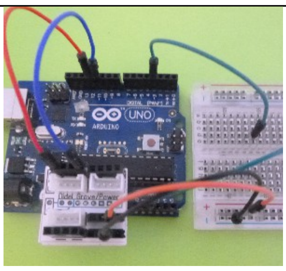
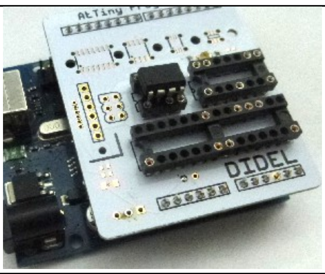
Programming AVR-AtTiny controllers under Arduino -- better title?

Programming an AVR microcontroller is done for years using AVR-studio and an ISP programmer. Now a large community have learned playing with Arduino and 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 is fun.
AVR has 8-pins and 14-pins microcontroller, similar to the Microchip PIC available for 15 years. An Arduino compatible board can be used to program these Tiny-8 and Tiny-14. The problem is one cannot, except for trivial applications, convert and download an Arduino program into a Tiny with 2k of memory, or even 8k. It is not only a question of memory space. Embedded processors have real time and power constraints a card interacting with a human does not have.
You cannot play the "Tiny" game if you do not have a good insight on how to program an Arduino not using the Arduino functions (digitalWrite, etc.). Several Arduino documents found on the Web go in that direction. We try to be more complete and as simple as possible.
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 ready to understand this document and write efficient "Tiny" programs.

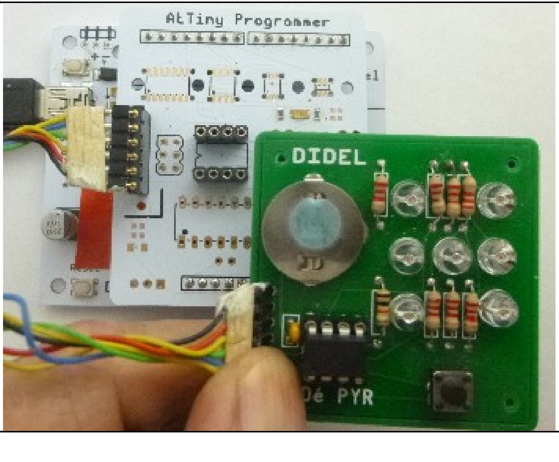
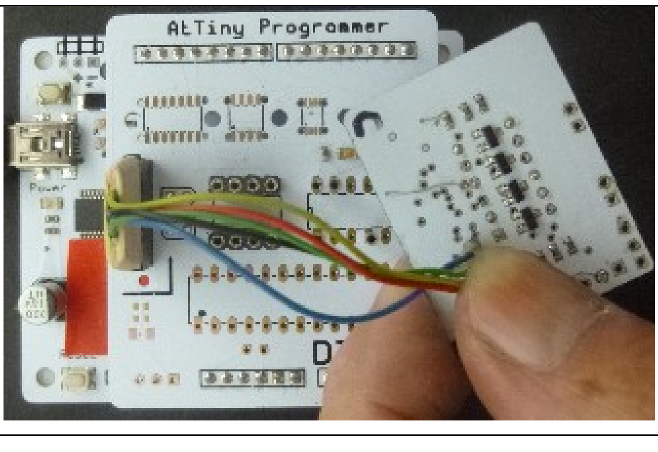
Ways of programming

We use the Didel AtTiny programmer available from different sources as a more stable and flexible solution to breadboard wiring. See www.didel.com/diduino/AtTinyProgrammer.pdf - soon (now doc 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 that could have used Arduino functions, but all these functions have been removed to do a "pure C" program that runs on the AVR328 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 situ programming avoid transferring the Tiny between the programmer and the board. Usually, many development loops are required: compile, program, see it is not yet good.
As example, the DePyr kit soldered by pupils can be reprogrammed with simpler C programs they understand. No connector, just push un the side during programming. Connector with a 1.27mm pitch is used for small SMD boards.

An important constraint due to ISP programming is that the signals of the application do not disturb the programmer that sends the CK and MOSI signals, and read the MISO from the AtTiny. Evaluate the sink and source currant on these pins when designing the schematic.
The AutoISP adapter of Matthias Nierarcher is a good way to be indenpendant of these wiring constraints if you use DIL packages – reference?

Developing the final software need many learning steps and test. We cannot tell you all in this document. Of course you know C and have written non trivial Arduino programs. You need to fully understand logical operations, be able to decode simple logic diagram and be willing to replace "of the shelf" libraries by sets of instructions that control directly the internal registers.
Do simple programs to test all features separately. Debugging of the end system is difficult, if even possible. See our tricks at the end of this document.

AVR Tiny-8 (13A, 25, 45, 85) and Tiny-14 (24, 44, 84)
These processor have different memory size. We do not care, we will show how to do a lot in 1k bytes and 64 8-bit variables (the Tiny13), but you have to loose all your 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 this.
We will work only with PORT bits. The layout of the Tiny8 and Tiny14 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
OC1A  MOSI   PA6  7|    |8   PA5  MISO OC1B
                    +----+
```

Selecting which pins to use as control signals for your application is not evident. Every pin, in addition to be a programmable input/output of a PORT, has possible connections, sometimes to the A/D converter, to a timer function, to an interrupt line, etc. Check data sheet if you have special constraints. 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.
Note: we will not use the Reset pin. It implies modifying the fuse and it is then difficult to program again (see ScratchMonkey for solutions).

Pins and Ports

Remember how to blink a led connected to PB4 pin3 on Tiny-8 or to PA0 pin 13 on Tiny-14, not using the Arduino facilities. If this is not familiar, this document is not yet for you.

Tiny-8
```
#define  bLed   4       // Bit 4 on PORTB
#define  LedOn   set   PORTB:bLed
#define  LedOff  clear PORTB:bLed
#define  LedToggle  PORTB ^= (1<<bLed)
```

Tiny-14
```
#define  bLed  0      // Bit 0 on PORTA
#define  LedOn   set   PORTB:bLed
#define  LedOff  clear PORTB:bLed
#define  LedToggle  PORTA ^= (1<<bLed)
```

Pins direction is usually defined in the Arduino setup, but we should not use the setup(), that trigger the loading of the heavy Arduino environment. C programs have only one main loop.

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

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

The Arduino Loop is replaced by a `while(1)`

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

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

Function `DelMs()` replace the delay() fucntion, that uses a timer and 200 bytes of code. We see later how to do it in 20 bytes.

Other conventions

As usual in C, constant are upper case, at least first letter
Fonctions start with an upper case, and are a verb that express the action
Variables are lower case. Upper cases are preferred to underline for subnames junction.
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.
Initial #defines must give names to the register bits. The associated pin number may depends on the package.
A led connected to AtTiny-8 pin 3 is bit 4 of PORTB, Use a `#define bLed 4 // PORTB`.

Delays

Repeating a no operation instruction (NOP) in a loop is the way to loose time. The processor runs at 16 Mhz, simple instructions takes ca 0.1 us. The compiler generates many instructions for a `for`, a `while`. One can study this with a scope. 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 bord 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
#define Calib1ms 900  // 1ms at 16 MHz
#include <avr/io.h>   // know names as PORTB DDRB ...
#define bLed 5   // portB pin 13
#define LedToggle PORTB ^= (1<<bLed)
void DelMs (int del) {  // max count 65535
   for (volatile int i=0; i<dm; i++) {
      for (volatile int j=0; j<Calib1ms; j++) {}
   }  // 900 loops for 1 second
}
```

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

You see the 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.

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

```
// Blink.ino    Tiny13/25/45/85  blink PB4 pin 3
#define Calib1ms 560   // 1ms at 10 MHz
#include <avr/io.h>
#define bLed 4   // PB4
#define LedToggle PORTB ^= (1<<bLed)
void DelMs (int dm) { // dm>0 delay in ms
   for (volatile int i=0; i<dm; i++) {
      for (volatile 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 TinyProgrammer, as documented on www.didel.com/diduino/AtTinyProg.pdf, the Tiny-8 you have selected is blinking on the TinyProgrammer led. For the Tiny24, the pin connected to the TinyProgrammer led is A0. Use the line `#define bLed 0   // PA0  and #define LedToggle PORTA ^= (1<<bLed)`. Of course, modify the setup: `DDRA |= (1<<bLed);  //PB0 out`

Note: If you want to be efficient developing and testing AtTiny programs, you should have two Arduino/Diduino. One is in the modes ***AVR ISP*** and ***Atmega328***, the other carry the programmer and the AtTiny (or the coble 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. Save all versions!

Get a key

Push-button have 1-5 ms bounces. One way to avoid bounces is to read them every 5ms.
This is frequently done when one wait for a key depressed.
As a test program, let us copy a button on the led. It shows how to test an input.
Button is active low, Led is active high

```
#define bButton 2  // PB2  Tiny8
#define ButtonOn (!(PORTB & (1<<bButton)))  // active low
#define bLed 4   // PB4
#define LedOn   set PORTB,bLed    // PORTB |= (1<<bLed) if you prefer
#define LedOff  clear  PORTB,bLed  // PORTB &= ~(1<<bLed)

#define Calib1ms 900
 void DelMs (int dm) { // dm>0 delay in ms
   for (volatile int i=0; i<dm; i++) {
      for (volatile int j=0; j<Calib1ms; j++) {}
   }
}
int main () {
  DDRB |= (1<<4);  //PB4 out  PB2 and others in
  while (1){
    DelMs (5);
    if (ButtonOn) { LedOn ; }
    else ( LedOff ; }
  }
}
```

Non blocking task
Non blocking mean that the execution will go through a small set of instructions, and continue with some memory of what happened. Next passage check again the evolution. The change in a button result from comparint the previous state and the ne one. It the scanning is too frequent, bounces will be seen. If not frequent, a short depress may not be seen. A flag is activated at the instant the key is depressed. This flag is cleared by the task waiting for the key.

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

Short and long actions on the button
One buttons can provide more than one bit of information. Recognizing short and long pushes just need one more variable and one flag.
Nicolas – cela vaut la peine de détailler? Ce serait un fichier annexe.

How many times depressed?
This is specially usefull at power-up. Depressing 1,2,3 times will start different programs.
See www.didel.com/diduino/DemosMultiples.pdf

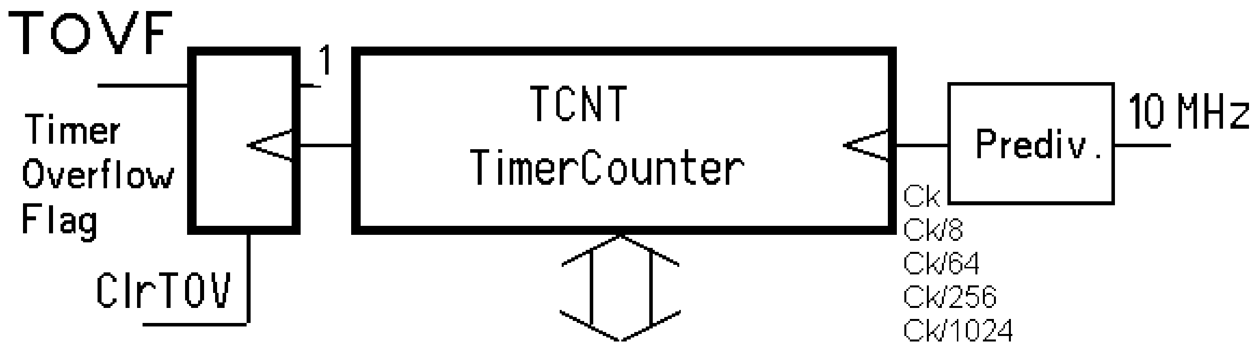
Interrupt driven button
It can be done, but if the button has bounces, is starts to be complicated and need to use a timer (Aduino examples use millis()). Doing one or several keys in a timer interrupt will be shown later.

Multi actions
One can do a lot, easy to program and to debug with a main loop of about 1ms duration and tasks that follow each other, lasting only the 10 lines of code.
We have seen how to activate a flag when a button is depressed. This flag is checked later.
Is 1ms too fast for the required 20ms bounces? just add a counter and handle the button every 20 cycles
    if (cbounce++>20) {cbounce=0; *-do instructions every 20ms -* }

Blink example
For instance, you can blink at any speed in a loop called every 1ms. Let us write a function that do
1) nothing if `flagBlink` is 0 (false)
2) blink n times at 5Hz when `FlagBlink` is true. The function clears `FlagBlink` when done.

```
#define DelOn 100  // max 255
#define DelOn 200  // max 255
 byte flagBlink
 void  Blink (byte nb) {  // called every 1ms  nb= 1,2,  0=256
    byte cntOn=1, cntOff=0;
    if (flagBlink) {
     if (nb > 0) {
      if (cntOff == 0) {
        LedOn;
        if (cntOn++ > DelOn) {
          cntOn=0; CntOff =1;
        }
      }  // end cntOn
      if (cntOn == 0) {
        LedOff
        if (cntOff++ > DelOff) {
          cntOff=1; CntOn =0;
        }
      }  // end cntOff
      nb--;
     }  end cnt blinks
     if (cntOff > 200) {} // attente
     flagBlink = 0;
   } // end blinks
}
```
-not tested.

Timer0 - 100 us loop
Timers are powerful, Our objective is not to study them, but use in an easy way, so we can also test on Arduino/Diduino before programming the Tiny. There are 2 timers. We use the 8-bit timer Tmr0 in its simplest form: generate an interrupt every 200 mictosecond. This will be our "clock" for the tasks to be done repetidively. Main program will be interrupted every 200 us, for 50 us. A regular interrupt is easy to handle There are a set of tasks executed bit at a time every 200 us, and the main programm that got only 70% of the power of the processur.



The interrupt needs to configure 2 registers, and then, when the timer overflow, it is reloaded with the time duration and the interrupt tasks are executed.

```
// Blink by interrupt
#include <avr/io.h>
int main ()  {
  // initialize leds, etc
  TCCR0A = 0; //default
  TCCR0B = 0b00000111;  // clk/1024
  TIMSK0 = 0b00000001;  // TOIE2
  sei();  // activate the interrupt
}
int cntLed;
ISR (TIMER0_OVF_vect)
{
  TCNT2 = 16;  // pour 1 kHz
  if (cntLed++ > 500) {
    cntLed = 0;
    Led1Toggle;
  }
pas testé
```

Pwm and pfm
PWM can be programmed on a timer, but the outputs must be on given pins. For controlling motors, is is so simple to prrogram PWM, and have the routine called every 200 microseconds by the timer interrupt. A resolution of 10 or 20 will be good enough.

SPI, S2, S1
The Tinys 8 and 14 have a SPI/i2c. Not so complicated if you can read the doc, and not follow the example of a guy how wanted to show he is smart and has programmed all possible functions.
As a rule, if the Tiny is a slave and must receive data as high speed burst, you need special hardware to keep-up. If the Tiny is a master that sends info or receive data at slow rates (kHz), routines in C are efficient and easy to control and debug.
We recommend to use simplified serial protocols S2 S1, specially designed to control a 4-digit display Afx4 or Dg4dig. A display controlled by a single output, and we recommend to use the Led output, is a powerfull debugging tool.

EEPROM
Saving data on EEPROM and get it back at power-up is frequently required with AtTiny applications.
One need 3 functions: setEeAddress, writeEe, readEe (autoincremented).

.		

Sleep
A powered Tiny can be put to sleep; the current will go to very low value and many pages explain how to go to the limits. Let's do it simple.
The problem is how to wake-up the processor. An interrupt can do this; it is rather easy to program, see ...DePyr (in french)

The watchdog is the other solution.

Watchdog

Debug
Led output could control a display. 1 line, 300 microsecond and 16 bits are displayd in hexadecimal or BCD form (Afx4/Dg4dig).