Introduction to the Atmel ATtiny15L

Jean-Michel FRIEDT, October 2001-December 10, 2001

1 Introduction

Altera developed a series of RISC microcontrollers based on an Harvard architecture (separate data and program memories) very similar to the Microchip PIC series. The program memory is based on a flash-type memory and can be safely erased and written to at least 1000 times.

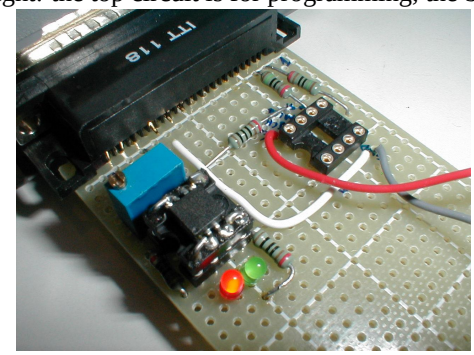
We have focused on the ATtiny15L which combines several attractive features:
- internal (low precision) RC oscillator which eliminates the need for an external quartz resonator
- low volume/PCB surface consumption as the microcontroller comes as an 8 pin DIP or SOIC package
- 4 10 bits analog to digital converters
- timers, watchdog, 4 general purpose I/O pins
- low power consumption in sleep mode.

This microcontroller is suitable for the development of simple intelligent data acquisition systems and more complex tasks such as an IP stack.

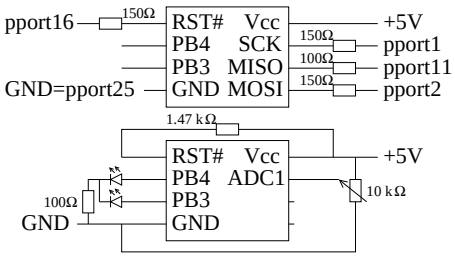
2 Development tools

The microcontroller needs a very minimal hardware setup for programming and running. A 4 resistors circuit connected to a PC parallel port is enough for transferring a program assembled on a linux based IBM compatible (equipped with a parallel printer port) computer to the ATtiny15L.

We used the following circuit for transferring programs from the PC to the microcontroller, using the linux-based software `uisp` (right: the top circuit is for programming, the bottom circuit for testing the programs presented later in this document):



PC parallel port	ATtiny15L
1	SCK
2	MOSI
11	MISO
16	RESET#



All connexions were made using 100 Ω or 150 Ω resistors for safety, although direct wire connexions should work fine. We used `uisp-1_0b_src.tar.gz` for generating the executable `uisp version 1.0b`. We will come back later to the actual command line for programming the ATtiny15L, as we should first see how to assemble a basic program. The right-most image on top shows the circuit for the programmer (top) and the test circuit used for executing the programs which will be presented later in this document (bottom).

If all goes well, the default output (in verbose `v=1` mode) of `uisp` is a message saying that "Atmel AVR similar to the AT90S1200 is found". With the enhanced verbose mode we have chosen here (`v=3`), the programmer should additionaly confirm that the Vendor code is 1e, the Part Family is 90 and the Part Number is 6. These values are used by Atmel for identifying an ATtiny15L.

We use the assembler `ava` for assembling our programs for the ATtiny15L. Although to this date (10/29/01) `ava` does not support the ATtiny15L, this microcontroller uses the same AVR mnemonics as the other Atmel RISC processors of the same series. We compiled `ava-0.3b` under linux and use the following script for assembling a program. Another option would be the use of `tpasm` which explicitly supports the ATtiny15L in version `tpasm version 1.0s`. Apart from the fact that `tpasm` compiles flawlessly under linux, we have not investigated further the use of this assembler which supports a wide range of processors, including the 6502, the 68HC11, the 16F84 and the Z80.

Assembling with `ava` is done in two steps, first generating an intermediate object file from the assembly program by running `ava program.s`, and then generating a Motorola S-record by executing `ava --motorola program.o`. This S-record is then written in the flash memory of the ATtiny15L using the following script:

```
#!/usr/bin/tcsh
if ( $# == 0 ) then                     # requires at least 1 argument
  echo $0 "filename[.asm]"
else
  set nom=$1                            # remove extension from name (if given)
  if ( `echo $1 | grep \.` != "" ) then
    set nom=`echo $1 | cut -d\. -f1`
  endif
  echo $nom "->" $nom".srec"            # input foo.asm outputs as foo.out
  ava $nom.s
  ava --motorola $nom.o
  mv a.out $nom.srec
  \rm $nom.o
endif
```

Script used for assembling the programs: name this file `asm` and call `./asm name_prog` for generatin the Motorola S-record `name_prog.s`.

```
pcfriedtj:/home/jmfriedt/avr/latex# ./prg ledadc.srec
Atmel AVR similar to the AT90S1200 is found.
Erasing device ...
Reinitializing device
Atmel AVR similar to the AT90S1200 is found.
Number of delay loops per 100 millisecond: 17657
port access granted, dropping permissions...
AVR Direct Parallel Access succeeded after 0 retries.
Vendor Code: 1e
Part Family: 90
Part Number: 6
```

```
Atmel AVR similar to the AT90S1200 is found.
Page Mode Enabled: No
FLASH Write Delay (t_wd_flash): 17444 us
EEPROM Write Delay (t_wd_eeprom): 8722 us
Auto-Uploading: flash
####
(total 102 bytes transfered in 1.87 s  min/avg/max = 57/55/57 bytes/s)
Atmel AVR similar to the AT90S1200 is found.
Auto-Verifing: flash
pcfriedtj:/home/jmfriedt/avr/latex#
```

Sample output as should be observed after a successfull programming of an ATtiny15L chip using the script `asm` described previously (executed on a 90 MHz Pentium based laptop.

```
#!/usr/bin/tcsh
if ( ‘id -u‘ != 0 ) then            # must be root to access hw (or suid ...)
  echo "Must be root to program AVR"
else
  if ( $# == 0 ) then               # requires at least 1 argument
    echo $0 "filename[.srec]"
  else
    set nom=$1                      # remove extension from name (if given)
    if ( ‘echo $1 | grep \.‘ != "" ) then
      set nom=‘echo $1 | cut -d\. -f1‘
      endif
    uisp -dapa -dno-poll -favr_endianbug --erase
    uisp -dapa -dno-poll -favr_endianbug -v=3 --upload if=$nom.srec
    uisp -dapa -dno-poll -favr_endianbug --verify if=$nom.srec
  endif
endif
```

Script for programming the ATtiny15L: first erase all words, then write (in verbose mode) the new program to flash memory, and finally verify that the program was correctly sent and written.

A first program example for blinking two LEDs connected between port B pins 3 and 4 and ground is given here, mainly to show what the assembly syntax looks like:

```
#arch ATtiny12
#include "avr.inc"

seg abs=0 flash.code  /*       ATtiny15L    */  /* AT90S1200  */
rjmp __init_          /* RESET              */
reti                  /* INT0               */
reti                  /* I/O pins           */  /* Timer0 OVF */
reti                  /* TIMER1, COMPA      */  /*  ANA COMP  */
reti               /* ATtiny15: TIMER1, OVF */
reti               /* ATtiny15: TIMER0, OVF */
reti               /* ATtiny15: EE_RDY      */
reti               /* ATtiny15: ANA_COMP    */
reti               /* ATtiny15: ADC         */

#define llop $10

__init_:                /* Initialize Hardware  */
ldi r16, $18      /* port B 3,4 as output */
out $17, r16      /* DDRB port = 0x17     */

forever:ldi r16,$10       /* switch LED (PB4) on */
out $18, r16
nop
```

```
ldi r17,llop
lo1b: ldi r16,llop      /* delay */
lo1a: dec r16
brne lo1a
nop
dec r17
brne lo1b
nop

ldi r16,$08       /* switch LED (PB3) on */
out $18, r16
nop

ldi r17,llop
lo2b: ldi r16,llop
lo2a: dec r16
brne lo2a
nop
dec r17
brne lo2b
nop

rjmp forever
```

Basic blinking-LEDs example

The first 9 words of the flash memory define the actions to be taken for the 9 interrupts that can occur: in our case, we ignore all interrupts (`reti`, return from interrupt) except the reset (and power on) which must jump to the beginning of our program.

The architecture of the ATtiny15L includes 30 general purpose 8 bits wide registers (which is to be used as RAM) and 64 I/O ports for controlling the peripherals (not all of them are used). This simple example only uses one register called `r16` and two ports, the port B Data Direction Register (for defining port B bits 3 and 4 as outputs) and port B Data Register (for defining if a LED is switched on – 1 – or off – 0). The ATtiny15L architecture does not allow the use of a stack by the program.

The picture showing the programmer attached to the parallel port also displays a circuit for testing the programs stored in the ATtiny15L: to the right (empty IC socket) is the programmer which connects the MOSI, MISO, SCK and RESET# lines to the parallel port, and to the left is the test circuit on which the RESET# line is connected to the +5 V power supply through an 1.6 kΩ resistor, and the two LEDs are connected to port B pins 3 and 4 on one side, and to groud through a 100 Ω resistor on the other side. The 10 kΩ potentiometer is connecter to the analog to digital converter 1 (ADC1), which can also be used as port B pin 2..

The following program reads the value on the potentiometer connected to ADC1 and changes the LED blinking rate so that the smaller the voltage on ADC1, the faster the LEDs blink:

```
#arch ATtiny12
#include "avr.inc"

seg abs=0 flash.code  /*       ATtiny15L    */  /* AT90S1200  */
rjmp __init_          /* RESET              */
reti                  /* INT0               */
reti                  /* I/O pins           */  /* Timer0 OVF */
reti                  /* TIMER1, COMPA      */  /*  ANA COMP  */
reti               /* ATtiny15: TIMER1, OVF */
reti               /* ATtiny15: TIMER0, OVF */
reti               /* ATtiny15: EE_RDY      */
reti               /* ATtiny15: ANA_COMP    */
reti               /* ATtiny15: ADC         */

__init_:                /* Initialize Hardware  */
ldi r16, $18      /* port B 3,4 as output */
out $17, r16      /* DDRB port = 0x17     */
nop
ldi r16, $21      /* ADMUX: Vcc=ref, right align (8 bits), ADC1/PB2 */
```

```
out $07, r16      /* ADMUX port = 0x07     */
nop
ldi r16, $80      /* ADCSR: ena ADC        */
out $06, r16      /* ADCSR port = 0x06     */
nop

forever:ldi r16,$10       /* switch LED (PB4) on */
out $18, r16
nop

sbi $06, $06      /* ADCSR: start conversion */
lopadc1:in r16,$06
cpi r16,$c0
breq lopadc1      /* while conversion is not finished, loop ... */
nop

in r17,$05        /* 8 bit value: read ADCH only*/
lo1b: ldi r16,$ff       /* delay loop, delay=value from ADC */
lo1a: dec r16
```

```
brne lo1a
nop
dec r17
brne lo1b
nop

ldi r16,$08       /* switch LED (PB3) on */
out $18, r16
nop

sbi $06, $06      /* ADCSR: start conversion */
lopadc2:in r16,$06
cpi r16,$c0
breq lopadc2      /* while conversion is not finished, loop ... */
nop

in r17,$05        /* 8 bit value: read ADCH only*/
lo2b: ldi r16,$ff       /* delay loop, delay=value from ADC */
lo2a: dec r16
brne lo2a
nop
dec r17
brne lo2b
nop

rjmp forever
```

Basic blinking-LEDs and analog to digital conversion example: the LEDs here blink with a delay period proportional to the voltage read on ADC1 (port B, pin 2).

3 Using the interrupts and the low-power mode

The first 9 words at the beginning of the flash memory of the ATtiny15L are instructions (`rjmp address`) to be executed when a given interrupt occurs. As an example, when a Reset occurs, the first instruction located at the first word is executed, and jumps to the beginning of the program to be run. We will here illustrate the use of two other interrupts: the ADC end of conversion interrupt and the timer overflow interrupt.

The two main advantages of using interrupts are:
- the program is not stuck in an infinite loop waiting for delays to be completed between each actions: the main loop can incorporate some time critical operation which must often be repeated and only sometimes spend some time on actions such as reading from the ADC
- power consumption reduction thanks to the use of the `sleep` instruction which will put the CPU in low power consumption mode until an interrupt is triggered.

```
#arch ATtiny12   /* R18 is used as a global var and should not be modified */
#include "avr.inc"

seg abs=0 flash.code  /*       ATtiny15L    */  /* AT90S1200  */
rjmp __init_          /* RESET              */
reti                  /* INT0               */
reti                  /* I/O pins           */  /* Timer0 OVF */
reti                  /* TIMER1, COMPA      */  /*  ANA COMP  */
reti               /* ATtiny15: TIMER1, OVF */
rjmp timer0        /* ATtiny15: TIMER0, OVF */
reti               /* ATtiny15: EE_RDY      */
reti               /* ATtiny15: ANA_COMP    */
rjmp adc_c         /* ATtiny15: ADC         */

__init_:                /* Initialize Hardware         */
ldi r16, $18      /* port B 3,4 as output        */
out $17, r16      /* DDRB port = 0x17            */
ldi r16, $10      /* port B init val (10 or 08)  */
out $18, r16      /* PORTB port = 0x10           */
ldi r16, $21      /* ADMUX: Vcc=ref, right align (8 bits), ADC1/PB2 */
out $07, r16      /* ADMUX port = 0x07           */
ldi r16, $88      /* ADCSR: ena ADC & interrupt  */
out $06, r16      /* ADCSR port = 0x06           */

ldi r16, $02
out $39, r16      /* TIMSK: timer 0 overflow interrupt */
out $32, r16       /* counter 0 value = initial value   */
ldi r16, $03
out $33, r16      /* TCCR0: speed timer0=CK/64 ; TCCR0 port = 0x33 */
sei

loop: sleep             /* either we have an ADC conversion running, or  */
rjmp loop         /* timer0 => we will wake up some day ...        */

adc_c: in r16,$05        /* read value from ADC   */
com r16           /* r16=$ff-r16           */
out $32,r16       /* counter 0 value = ADC */
ldi r16, $03
out $33, r16      /* TCCR0: speed of timer0=CK ; TCCR0 port = 0x33 */
reti

timer0: dec r18           /* delay due to ADC is amplified 256 times    */
brne theend       /* only change the LEDs once every 256 calls  */
ldi r18,$ff
in r16,$18        /* read port B                                */
ldi r17,$18       /* XOR avec bits de portB 3,4 => inversion    */
eor r16,r17       /* r16=r16 XOR r17                            */
out $18, r16      /* switch LEDs on/off                         */
theend: sbi $06,$06       /* start a new conversion to know delay value */
ldi r16, $00
out $33, r16      /* stop timer for the moment (until ADC)      */
/* ldi r16,$0f;out $32,r16;ldi r16,$03;out $33,r16    <- NO ADC */
reti                                        /* ie for cst delay */
```

The LEDs still blink with a period proportional to the voltage read on ADC1, but this time the empty loops are replaced by the `sleep` instruction for entering low-power mode. The processor is waken up either by a timer0 interrupt, or by an ADC conversion complete interrupt.

As an example of the second point, the program `ledadc.s` presented earlier leads to a power consumption (after disconnecting the LEDs which sink most of the current otherwise) of 5.75_5 mA, slightly greater than the program `ledadcirq.s` we just presented which leads to a power consumption of 5.64 mA.

4 RS232 communication

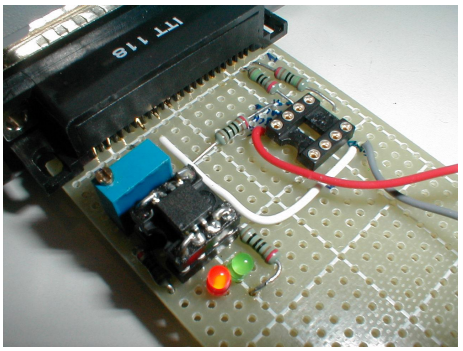
4.1 Polled UART emulation software

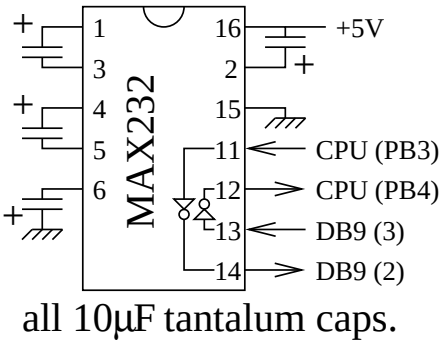
The ATtiny15L is not provided with an UART: a software emulation of the asynchronous communication protocol is thus required. An additional complication is that the oscillator of the ATtiny15L is not calibrated at the factory. A wide range of frequency can thus be observed on the default setup. Two options are available: calibrate the internal oscillator using the `OSCCAL` calibration register ($31), or modify the software delay in the UART emulation routine. Due to the lack of calibration routine for the former solution, we opted as a short term solution to the latter approach. Altough not satisfactory, it appeared suitable for the two microcontrollers available during this test in order to achieve reliable 1200 baud rate. Achieving 2400 baud rate communication required a theoretical calculation of the delay duration as presented later in this document.

We based our software UART emulation on the Atmel AVR305 application note, slightly modified in order to adapt the syntax to the assembler used here, and with a new delay value suitable for the ATtiny15L oscillator.

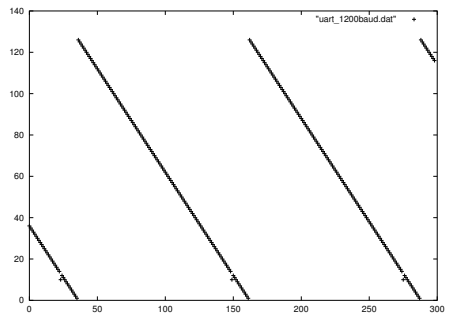
```
;**** A P P L I C A T I O N   N O T E   A V R 3 0 5 ************************
;* Half Duplex Interrupt Driven Software UART V1.1 (97.08.27)

#arch ATtiny12
#include "avr.inc"

#define RxD  04 ;Receive pin is PB4
#define TxD  03 ;Transmit pin is PB3

#define bitcnt  R16 ;bit counter
#define temp  R17 ;temporary storage register

#define Txbyte  R18 ;Data to be transmitted
#define Rxbyte  R19 ;Received data

        seg abs=0 flash.code  /*       ATtiny15L    */  /* AT90S1200  */
        rjmp reset            /* RESET              */
        reti                  /* INT0               */
        reti                  /* I/O pins           */  /* Timer0 OVF */
        reti                  /* TIMER1, COMPA      */  /*  ANA COMP  */
        reti               /* ATtiny15: TIMER1, OVF* /
        reti               /* ATtiny15: TIMER0, OVF */
        reti               /* ATtiny15: EE_RDY      */
        reti               /* ATtiny15: ANA_COMP    */
        reti               /* ATtiny15: ADC         */

; "putchar": transmits the byte stored in the "Txbyte" register
; the number of stop bits used is set with the sb constant
#define sb 1 ;Number of stop bits (1, 2, ...)

putchar: ldi bitcnt,9+sb ;1+8+sb (sb is # of stop bits)
com Txbyte ;Invert everything
sec ;Start bit

putchar0: brcc putchar1 ;If carry set
cbi PORTB,TxD ;    send a '0'
rjmp putchar2 ;else

putchar1: sbi PORTB,TxD ;    send a '1'
nop

putchar2: rcall UART_delay ;One bit delay
rcall UART_delay

lsr Txbyte ;Get next bit
dec bitcnt ;If not all bit sent
brne putchar0 ;   send next else
ret ;   return

; "getchar": receives one byte and returns it in the "Rxbyte" register

getchar: ldi  bitcnt,9 ;8 data bit + 1 stop bit

getchar1: sbic  PINB3,RxD ;Wait for start bit
rjmp  getchar1 ; HERE WE COULD MEASURE START BIT LENGTH
                    ; AND AUTOSET BAUDRATE ... (XMIT & RCV)
rcall UART_delay ;0.5 bit delay

getchar2: rcall UART_delay ;1 bit delay
rcall UART_delay

clc ;clear carry
sbic  PINB3,RxD ;if RX pin high
sec ;

dec  bitcnt ;If bit is stop bit
breq  getchar3 ;   return else
ror  Rxbyte ;   shift bit into Rxbyte
rjmp  getchar2 ;   go get next
getchar3: ret

;* "UART_delay"
;* This delay subroutine generates the required delay between the bits when
;* transmitting and receiving bytes. The total execution time is set by the
;* constant "b":
;
; 1 MHz crystal: ;   9600 bps - b=14 ;  19200 bps - b=5 ;  28800 bps - b=2
; 2 MHz crystal: ;  19200 bps - b=14 ;  28800 bps - b=8 ;  57600 bps - b=2
; 4 MHz crystal: ;  19200 bps - b=31 ;  28800 bps - b=19 ;  57600 bps - b=8

#define  b 130 ; 1200 bps for the ATtiny15L -- should be 128 (cf text)
; #define b 66 ; 2400 bps for the ATtiny15L -- calculated val works ok

UART_delay: ldi temp,b
UART_delay1: dec temp
brne UART_delay1
ret

reset: sbi PORTB,TxD ; *** Program Exec Starts Here
sbi DDRB,TxD ; Init port pins: DDRB=$17

forever: ldi R20,$7e
forever1: ; rcall getchar
; mov Txbyte,Rxbyte
mov Txbyte,R20
rcall putchar ;Echo received char
dec     R20
brne forever1
rjmp forever
```

Software UART emulation based on the Atmel AVR305 application note. This example simply sends bytes from 0x7E to 0x00 to the serial port at 1200 bauds. Only the byte sending subprograms are used, although the application note also provides the routines for byte receiving.





Additions to the previously described test circuit: the MAX232 is required to shift the TTL compatible signals to ±12 V voltages required for RS232 communication. The LEDs connected to PB3 and PB4 are left on the circuit as communication indocators.

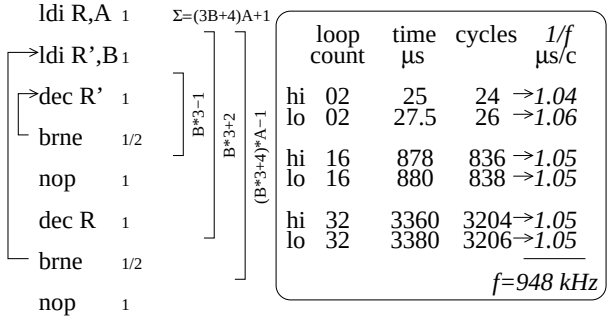


36($24)	17($11)	124($7c)	105($69)	86($56)	67($43)
35($23)	16($10)	123($7b)	104($68)	85($55)	66($42)
34($22)	15($f)	122($7a)	103($67)	84($54)	65($41)
33($21)	14($e)	121($79)	102($66)	83($53)	64($40)
32($20)	10($a)	120($78)	101($65)	82($52)	63($3f)
31($1f)	12($c)	119($77)	100($64)	81($51)	62($3e)
30($1e)	11($b)	118($76)	99($63)	80($50)	61($3d)
29($1d)	10($a)	117($75)	98($62)	79($4f)	60($3c)
28($1c)	9($9)	116($74)	97($61)	78($4e)	59($3b)
27($1b)	8($8)	115($73)	96($60)	77($4d)	58($3a)
26($1a)	7($7)	114($72)	95($5f)	76($4c)	57($39)
25($19)	6($6)	113($71)	94($5e)	75($4b)	56($38)
24($18)	5($5)	112($70)	93($5d)	74($4a)	55($37)
23($17)	4($4)	111($6f)	92($5c)	73($49)	54($36)
22($16)	3($3)	110($6e)	91($5b)	72($48)	53($35)
21($15)	2($2)	109($6d)	90($5a)	71($47)	52($34)
20($14)	1($1)	108($6c)	89($59)	70($46)	51($33)
19($13)	126($7e)	107($6b)	88($58)	69($45)	
18($12)	125($7d)	106($6a)	87($57)	68($44)	

Output example of the program described at the beginning of this section as monitored by a linux running laptop (1200 bauds communication, N81).

The result displayed here was obtained after a few random guesses as to the value of b. A better approach is to calibrate the oscillation frequency of the ATtiny15L being used and predict, once the oscillator frequency is known (and considered constant), the value of b.

Let us first see how the measurement of the oscillator frequency can done. We consider the following part of the first example program we saw, which aims at making LEDs blink (the left-most column is the program sample, the middle column the cycle count, and the table to the right shows the delailed comparison between the execution time and the number of cycles for various bound values).



We here display two loops with boundaries values A (outer loop) and B (inner loop) to 0. This structure is similar to the delay loop used in the first example. The table to the right displays the calculated number of cycles for various values of A=B (A=B=2, 16 and 32 in the tests we performed). The third column from the left shows the measured durations (as seen on the oscilloscope whose probes were connected to the output pin PB3 of the ATtiny15L being analyzed). Once we know the number of cycles and the duration it takes to execute the loop, we can deduce the internal oscillator frequency (which is equal to one cycle period). In our case, we can see that all measurements are in close agreement with $f_{osc} \simeq 948$ kHz which is in the range 0.8-1.6 MHz as stated by the manufacturer. The only tricky part of calculating the number of cycles required for executing the loops is in including the duration of an `brne` instruction. The `brne` instruction takes two cycles to execute if the jump occurs (the result of the last operation was non-zero), and only one if no jump has to occur (the result of the last instruction was zero).

From this result ($f_{osc} \simeq 948$ kHz) we can come back to the `b=130` delay value chosen in the last example (software UART emulation for 1200 bauds communication). 1200 bauds transfer means each byte/character transfer requires a signal to update its state 1200 times per second (in the case of our N8-1 protocol, each byte takes 10 bits to be transferred, hence a transfer rate of 120 bytes/seconds). At 1200 baud, we thus require 833 μs/bit (including the start and stop bits, since $\frac{10^6}{1200} = 833$). By calculating the number of cycles required for running the `putchar` routine of the last example, we can predit the delay actually obtained (the only difficulty here is to remember that `bccr` is executed in one cycle if no jump occurs, and in 2 cycles if the jump to `putchar1` occurs). The `UART_delay` routine runs a loop from `b` to 0 and takes a total of $3 \times b + 4$ cycles to run, including the `ret` instruction. The total program part starting at `putchar0` and ending at the `ret` instruction of `putchar` takes an average of $(3 \times b + 4) \times 2 + 14$ (the last additional coefficient, 14, is an average of the value 15 obtained if carry set and 13 if not at `putchar0`). Hence, for $b = 130$, we see that the delay between two transmitted bits is 802 cycles which at a clock frequency of 948 kHz take 846 μs to be executed. This result is quite close to the expected 833 μs/bit (the difference, 13 μs, is only 1.5% of the total duration, which well withing the time interval displayed by most UART of microcontrollers for baud rates which are not exactly a multiple of their clock frequency). We can also predict that an optimum value of `b` would have been 128 in order to obtain a total subroutine duration between each transmitted bits of 790 cycles (=833.45 μs).

We have until now been able to measure the frequency of the internal oscillator of the ATtiny15L and deduce from it the delay loop range in order to achieve the required delay for the UART software implementation. However, such a measurement has to be made for each new microcontroller to be used. Two option are available: using the `OSCCAL`) register in order to calibrate the oscillator frequency to a given value for all microcontrollers (ideally 1.6 MHz, the highest clock frequency usable on this device), or adapting automatically the delay to the communication speed of the PC to which the microcontroller is connected. The latter approach not only allows for compensating for any drift of the internal oscillator (from chip to chip or due to temperature or supply voltage fluctuations), but also allows automatic adaptation to a broad range of communication baud rates (within the measurement accuracy and the boundary value within the 0 to 255 range to be stored in an 8 bit register).

4.2 Auto baud-rate adaptation UART software

Considering we know the reception baud rate (as sent by the PC to the microcontroller), we are able to measure the width of the start bit of the RS232 frame, and set the calibration counter accordingly (as done for example by the Hitachi H8/3048 bootstrap mode routine). We are then able to easily adapt the baud rate of the microcontroller to that of the PC by setting the variable `b`, rename `count` in this example, defining the delay in the UART emulation to a value compatible with the delay observed on the start bit of an RS232 frame.

Measuring the duration of the start bit requires the first data bit to be high (since the start bit is defined as low), *i.e.* the first datum sent must be an odd number. In order to avoid any risk of confusion when measuring the duration of the (active low) start bit, we first send the value $FF (which is transmitted as a low-level start bit followed by all-high bits defining the $FF value and the stip bit) from the PC to the microcontroller, and then listen to the values sent by the microcontroller.

We have tested this algorithm with 1200 baud and 2400 baud communication: the microcontroller was able to automatically adapt its transmission speed to the initial baud rate of the transmission of the $FF byte. Getting the right baud rate measurement sometimes requires several attempts, most certainly due to variations in the initial settings of the RS232 port (a proper C program on the PC side should make sure the serial port is at the high level when the microcontroller is switched on). The result for the delay value (`count` variable) is $65 or $66 (=102 decimal) for 1200 baud transmission rate, and $32 (=50 decimal) for 2400 baud transmission rate (which, since the delay time of this last example is close to $4 \times$ *count* since we introduced a `nop` instruction in the loop, and the clock frequency is 948 kHz, means we use a delay of 2×430 μs and 2×210 μs, quite close to the expected $\frac{10^6}{1200} = 833$ and $\frac{10^6}{2400} = 416$ μs. 600 baud transmission rate could not be obtained using this program because the counter loop is getting greater than 256 (a delay of $\frac{10^6}{600} = 1666$ μs

requires 1580 cycles to be executed, *i.e.* a 2`counter` value of $395 > 256$ during the measurement of the delay loop [1]).

```
;**** A P P L I C A T I O N   N O T E   A V R 3 0 5 ************************
;* Half Duplex Interrupt Driven Software UART V1.1 (97.08.27)

#arch ATtiny12
#include "avr.inc"

#define RxD  04 ;Receive pin is PB4
#define TxD  03 ;Transmit pin is PB3

#define bitcnt  R16 ;bit counter
#define temp  R17 ;temporary storage register

#define Txbyte  R18 ;Data to be transmitted
#define Rxbyte  R19 ;Received data

#define count  R21 ;Received data

        seg abs=0 flash.code  /*       ATtiny15L    */  /* AT90S1200  */
        rjmp reset            /* RESET              */
        reti                  /* INT0               */
        reti                  /* I/O pins           */  /* Timer0 OVF */
        reti                  /* TIMER1, COMPA      */  /*  ANA COMP  */
        reti              /* ATtiny15: TIMER1, OVF* /
        reti              /* ATtiny15: TIMER0, OVF */
        reti              /* ATtiny15: EE_RDY      */
        reti              /* ATtiny15: ANA_COMP    */
        reti              /* ATtiny15: ADC         */

; "putchar": transmits the byte stored in the "Txbyte" register
; the number of stop bits used is set with the sb constant
#define sb 1 ;Number of stop bits (1, 2, ...)

putchar: ldi bitcnt,9+sb ;1+8+sb (sb is # of stop bits)
com Txbyte ;Invert everything
sec ;Start bit

putchar0: brcc putchar1 ;If carry set
cbi PORTB,TxD ;    send a '0'
rjmp putchar2 ;else

putchar1: sbi PORTB,TxD ;    send a '1'
nop

putchar2: rcall UART_delay ;One bit delay
rcall UART_delay

lsr Txbyte ;Get next bit
dec bitcnt ;If not all bit sent
brne putchar0 ;   send next else
ret ;   return
```

```
; "getchar": receives one byte and returns it in the "Rxbyte" register
getchar: ldi  bitcnt,9 ;8 data bit + 1 stop bit
ldi count,0
getchar1: sbic  PINB,RxD ;Wait for start bit (while RxD=hi)
rjmp  getchar1
sbi PORTB,TxD ; *** Program Exec Starts Here

mycount: inc count     ;1  ; AUTOSET BAUDRATE ... (XMIT & RCV)
sbis PINB,RxD    ;1/2; the START bit is still occuring ...
rjmp mycount     ;2  ; here, delay(stop bit)=count*4
lsr count       ;1  ; and divide by 2

rjmp  get_JMF   ; 0.5 bit delay (already waited 1 delay)

getchar2: rcall UART_delay ;1 bit delay
get_JMF: rcall UART_delay

clc ;clear carry
sbic  PINB,RxD ;if RX pin high
sec ;

dec  bitcnt ;If bit is stop bit
breq  getchar3 ;   return else
ror  Rxbyte ;   shift bit into Rxbyte
rjmp  getchar2 ;   go get next
getchar3: ret

UART_delay: mov temp,count   ; ldi cycles=mov cycles JMF
UART_delay1: nop      ; added so that delay \propto count*4
dec temp
brne UART_delay1
ret

reset: cbi PORTB,TxD ; *** Program Exec Starts Here
sbi DDRB,TxD ; Init port pins: DDRB=$17
cbi DDRB,RxD ;  |

rcall getchar
mov Txbyte,count   ; send count val: $65-$66(=102)=1200 bps
                                    ;                    $32(=50)=2400 bps
rcall putchar        ;Echo received char
forever: ldi R20,$7e
forever1: ; mov Txbyte,Rxbyte
mov Txbyte,R20
rcall putchar ;Echo received char
dec     R20
brne forever1
rjmp forever
```

UART software emulation with automatic baud-rate detection. The baud rate is adapted by measuring the duration of the start bit, which allows for compensation in the variations of the communication baud rate as well as to variations in the microcontroller internal clock frequency.

```
#include "rs232.h"

void read_osc(int fd)
{unsigned char buf;int i;
 buf=0xff;write(fd,&buf,1);                          /* start with 'FF' */
 printf("sent 0xff\n");
 read(fd,&buf,1);printf ("%u($%x)\n",buf&0x000000FF,buf&0x000000FF);
 printf ("Press enter to continue\n");fflush(stdout);
 scanf("%c",&buf);
 while (1) {read(fd,&buf,1);
   printf ("%u($%x)\n",buf&0x000000FF,buf&0x000000FF);
   fflush(stdout);}
}

int main(int argc,char **argv)
{int fd;
 fd=init_rs232();
 read_osc(fd);
 /* free_rs232(); */
 return(0);
}
```

C program used for testing the ATtiny15L program presented previously. This program sends a $FF byte to the microcontroller (for bit duration measurement), reads the delay value measured and, after the user hits the return key, reads the bytes sent by the microcontroller (which should be a decreasing count).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>                 /* declaration of bzero() */
#include <fcntl.h>
#include <termios.h>
int init_rs232();
void free_rs232();
void sendcmd(int,char*);
struct termios oldtio,newtio;

#define BAUDRATE B4800
// #define BAUDRATE B1200
// #define BAUDRATE B2400
// #define BAUDRATE B19200
#define HC11DEVICE "/dev/ttyS0"
```

`rs232.h` RS232 communication definitions (header file used in the C program presented previously). Modifiy the definition of `BAUDRATE` in order to test various baud rates.

These automatic baud rate detection programs can be used to roughly calibrate the internal oscillator of the microcontroller since the start bit duration for a given baud rate is theoretically known ($T_{start} = 1/baud\_rate$ s).

These programs demonstrate the ability of the automatic baud rate software UART to be designed, but lack safety checks such as averaging the measurement over several start bit delays or initial RS232 line level dependency. Such features should be added, depending on available code space, for practical applications.

[1] which is only later divided by 2 – an option would be to double the duration of the measurement loop, but the loss of time measurement resolution might affect the stability of the transmission at the higher baud rates which require a high resolution.