

1. Introduction.

The goal of the project is to manage a MCU AVR "AT tiny 85/84" using C/C++ code without using external libraries and avoiding the use of assembler coding (i.e.: very minimal use of assembler code). The full development should cover all aspects of the features these MCUs cover.

The source clock used to clock the MCU will be internal clock. This allows to use all the I/O pins the MCU has. I.E.: AT tiny 84 will use 8Mhz internal clock, AT tiny 85 will use 16 Mhz internal clock.

Although, at now, the code seems able to fit inside smaller tiny MCUs, the development will target "tiny 84/85" MCUs.

The actual state of art is very minimal; it try to cover timing and SPI communication (TWI should be better).

This document is an overview of the developed software and it tries to discuss about the development approach. It contains general indications. All the developed modules have methods commented in the relevant include files and their use is/will be demonstrated with code examples.

1.1. Pre-requirements.

All the software have been and will be developed under Linux, but (I think) is possible to use all the software relevant to the MCUs also under other operating systems such as MS Windows or Mac platforms. This will not be always true for any software tools which have been developed specifically for PC, to get them run under other operating systems than Linux they will have to be modified.

The minimal requirements to use this software are the following items:

- A PC or something similar :)
- The avr-gcc compiler and tools (this software may be found in the Arduino suite).
- Something to program the MCU flash. You may use Atmel specific products. I use the SW ArduinoISP running on an Arduino Atmega 2560 board. The makefiles in this project has a rule (rom) to program MCU flash using avr-dude with the setting needing to use ArduinoISP.

2. SW Modules.

The SW is managed in a project directory that contains all the files needed to compile some small applications. For now, applications, C/C++ libraries and their include-files are together in the ./ path.

The include-files which describe the hardware, the data-types and some bit operations are in the ./include subdirectory; in the ./include/reg subdirectory are all the include-files describing the hardware registers and relevant settings.

Most of the macros which describe registers and settings have names fitting those in the Atmel MCU documentation.

The makefile to make the applications are in subdirectories, located under the ./ path, named as the application which will be compiled using the make program inside them.

2.1. HW initialization.

The only function used to pre-initialize the whole MCU is **inithw()**. This function is developed as inline code, it's in the include-file inithw.h.

2.2. Timer module (attimer.cpp)

The attimer.cpp C++ code is the time module. It manages a timer-counter to generate a 64 bits time expressed in usec with the resolution specified by the TTICK_us macro (attimerdefs.h). The only way to change the resolution is to modify the timer-counter clock-divisor (TCLK_DIV macro in the file attimerdefs.h) taking in account that the maximum tick the TTICK_us macro may represent is 1 usec.

2.2.1. Main methods.

The main methods this module exposes are:

begin	This method initializes the timer. The begin method of the first instance of the ATTimer class activates the HW.
reset	This method restarts the timer. The time count restarts from 0. The reset method of the first instance of the ATTimer class resets the HW.
baseTime	This method returns the time elapsed since the begin (or reset) method initialized the HW. This time is expressed in usec (64 bits).
udelay	This method waits a numbers of usec. The wait-time shall be rounded taking in account the value of the TTICK_us macro
time	This method returns the time elapsed since the begin (or reset) method was called. This time is expressed in usec (64 bits). Calling this method for the first instance of the ATTimer class has the same result of the method baseTime().

E.G.: This code is “real-time” if the “Do something” actions have a duration of maximum 200 usec.

```
#include "include/inithw.h"
#include "attimer.h"
#include "include/h_types.h"

ATTimer timer();
void main(void) {
    uint64_t t0, t1, t2, t3;

    inithw();
    timer.begin();

    t3=t2=t1=baseTime();
    for(;;) {
        t0=timer.baseTime();
```

```

        if (t0-t1>=_Sec(1)) {
            //Do something every second!
            t1=timer.baseTime();
        }

        if (t0-t2>=_mSec(100)) {
            //Do something every 100 msec!
            t2=timer.baseTime();
        }

        if (t0-t3>=_uSec(300)) {
            //Do something every 300 usec!
            t3=timer.baseTime();
        }

        udelay(_uSec(100));
    }
}

```

2.2.2.Remarks.

For now this module uses the counter timer 1 as default. This behavior should be modified because tiny 84 uses timer counter 1 for the PWM functionalities unlike tiny 85 which uses the timer counter 0. It's possible to make the module uses the counter timer 0 specifying `-DWATCH_IS_TIMER0` when compiling the module attimer.

It's possible to have more instances of the `ATTimer` class, but only the first manages the HW behavior.

The reset method of the first instance doesn't reset all the `ATTimer` instances, then, when the method reset of the first instance is called, it need to call the method reset for all the other instances.

2.3. SPI Module (spi.cpp)

This module allows SPI master and slave communications. For now only mode 0 is implemented (I.E.: Data are sampled on the clock signal (SCK) rising edge and the clock signal is active-high).

This module has the following characteristics:

- The use of two circular buffers: one to receive data and one to keep the data to transmit.
- The use of SPI counter overflow interrupt to receive data when slave mode is selected.
- The use of polling to manage master communications.
- The capability to disconnect the SO (slave output) pin by means of a specific method when in slave mode.
- The capability to manage a protocol which allows to recognize the data validity (IE: The protocol is able to understand if 0xFF or 0x00 are data or not) and which slave is required to reply the master answers.

2.3.1. Base mains methods.

The base mains methods the SPI module exposes are:

begin	This method initializes the module and specifies the communication mode and the communication buffers.
enabled	This method sets the direction of the pins needed for the required SPI communication mode and activates the interrupt for the slave receiver if “not master” has been specified by the begin method (default).
outputEnable	This method disables/enables the output pin.
getRcvdData	This method takes bytes (and types) from the receiver buffer.
getRcvdByte	This method takes a single byte from the receiver buffer.
getRcvdDataSize	This method returns the number of byte in the receiver buffer.
setData2Xmit	This method puts bytes (and types) in the transmission buffer.
setByte2Xmit	This method puts a single byte in the transmission buffer.
get2XmitDataSize	This method returns the number of byte in the transmission buffer.
sendMaster	This method is used from the master to send data that are kept into the transmission buffer. When the method is called it may or may not receive data from the slave ("don't receive data" means don't keep incoming data into the receive buffer), this is the default. To receive data you may call the method sendMaster with or without data in the transmission buffer and specifying the length of the data you want receive.

2.3.2. An idea of protocol.

The software in this project tries to implement a communication protocol based on the idea to use the character 0x1B as prefix. This allows the programs to avoid to read data due to link absence (I.E. 0xFF or 0x00) and allows a master to communicate via software a code act to enable the correct slave that recognize such a code.

This protocol uses the byte 0x1B as prefix for the bytes 0xFF, 0x00 and 0x1B. All other chars prefixed by the 0x1B will be kept as they are (without the 0x1B code), but their code will be stored as a slave-code.

For now also the code 0xAA shall be prefixed by the 0x1B code because it's intended as slave presence, but I think I will remove this feature.

2.3.3. Method for the protocol.

The method (in addition to what above specified) implemented to use the protocol are the following:

enhancedSetData2Xmit	This method puts bytes (and types) in the transmission buffer.
enhancedGetRcvdData	This method takes bytes (and types) from the receiver buffer.
masterSetActiveSlave2Xmit	This method sets the activation code that a master will be sent to

	activate a slave.
activeSlave	This method reads the eventual slave activation code the master has sent.
setOutput	This method allows the slave to disconnect its output from the communication buffer.

2.3.4. Remarks.

I apologize, but this module is not capable of very high communication speed. In slave mode the limits is due to the ISR preamble load; in master mode the limits is due to the actual SCLK management (polling).

In slave mode the SW may receive data sent by the master at speed till 1 Mbit/sec (and over I think), but it need a delay of at least 10 usec after each received byte. This means the master may transmit at a speed of 1 Mbit/sec, but it has to insert a delay of at least 10 usec before sending another character.

2.4. Circular buffer module (circbuff.cpp)

This module implements a circular buffer class.

2.5. Some standard C functions and not. (fnclib.c)

The file fnclib.c implements some standard C functions prefixed by the underscore character “_”: `_strcpy`, `_strlen`, `_strcat`, `_memcpy`, `_memset`.

For now is possible to declare inline `_memcpy` and `_memset` using the declarations `#define __FNC_INLINE_MEMCPY` and `#define __FNC_INLINE_MEMSET` before the declaration `#include "fnclib.h"`.

An other function implemented in this module is `strtoll`. This function converts a string in its corresponding 64 bits integer value. It also understands standard C basis specifications IE: `0xnnnn` (for hexadecimal) and `0nnnn` (for octals), moreover this function understands the format `"*Bnnnnnn"` where B indicates a numerical basis IE: `"*21011"` will be converted in the decimal value 11; the value of the B field may be also a letter where "A" is interpreted as 10 basis and so on till "Z" that is interpreted as 36 basis. (see the include file `fnclib.h`)

2.6. Include files.

2.6.1. MCU Clock settings.

The include-file `timing.h` describes the MCU-clock settings. It's in the subdirectory `./include/reg`. The only way to specify different clock is to modify its contents.

Pay attention: the modifications of the timing in the file `timing.h` are not directly reflected on the MCU clock that shall be configured in the fuse-memory (see: AVR manuals).

The actual software configuration assumes the following fuse-memory configurations.

AT Tiny 85 16 MHz lfuse:0xE1 hfuse:0xdf efuse:0xff

AT Tiny 84 8 MHz lfuse:0xE2 hfuse:0xdf efuse:0xff

2.6.2.HW registers.

The HW registers and relevant settings are described in the include files contained in the directory `./include/reg`.

`regdport.h` digital I/O ports.

`regtiming.h` timer counters.

`regusi.h` USI (Universal Serial Interface)

`regadc.h` ADC (Analog to Digital converters)¹

2.6.3.Other include files.

Some include files are in the directory `./include`.

`bits.h`

`h_types.h`

`inithw.h`

3. Examples.

3.1. Make rules.

#make all

This compiles the code.

#make rom

This sends the code in the flash memory of the MCU (using as programmer an “Arduino as ISP”).

#make clock

This sets the MCU fuses to have the required timing (using as programmer an “Arduino as ISP”).

3.2. sbk85.cpp

The module `sbk85.cpp` is an example (for now the only. I apologize!!!). It starts with an output signal that blinks with period of 2 seconds high and 2 seconds low. On the AT Tiny 85 is used the pin B4, on the AT Tiny 84 is used the pin B2. The software is configured as SPI slave and is able to receive

¹ To be developed – (The next step I hope do develop)

simple commands that are the following:

t#	<p>This command returns information about the MCU timing.</p> <p>t xx yy zzzz OK (each char is a byte)</p> <p>Where: xx is the MCU tick in tenth of nanosecond, yy is the timer resolution in microseconds, zzzz is the time, in microseconds, elapsed from the timer activation (the low 4 byte – the time is 8 byte).</p> <p>The MCU generates the output data using this code: spi.setData2Xmit((uint16_t)TICK_ns_10th); spi.setData2Xmit((uint16_t)timer.uTick()); spi.setData2Xmit((uint32_t)timer.baseTime());</p>
blttnnnnn#	<p>This command obtains the signal specified by l blinks tt times with a period of nnnnn milliseconds.</p> <p>l may be 1 or 0 for the AT Tiny 84 (for the AT Tiny 85 is the same). On the tiny 84 1 means to send the signal on the pin B0, 0 means to send the signal on the pin B2. On the Tiny 85 is used only the pin B4.</p>
lnnnnnnnn#	<p>This command changes the time for which the signal B2 (AT Tiny 84) - or B4 (AT Tiny 85) - is alternatively high or low.</p>

If the received data are not correct the SW replies with 3 bytes “x?y” where x is the first byte received as command and y is an error code, ? Is ? :)

3.3. How to make the example.

To compile the example go in the directory project/sblk85 and type:

```
#make all <enter>
```

To upload the MCU you may use:

```
#make rom <enter>
```

(see the paragraph Pre-requirements)

To compile for the AT Tiny 84 you have to set the CPU before to execute make:

```
#export CPU=tiny84
```

then:

```
#make all
```

If you want the MCU uses the timer-counter 0 instead the timer counter 1 you have to specify this:

```
#export DEFS=-DWATCH_IS_TIMER0
```

then:

#make all

When you change configurations you have to remember to execute:

#make clean

before recompile the software.