CTmeasure - ATMega328p

Author: nmt@nt-com.org

Abstract

This document describes the CTmeasure project ported to the 8-bit ATMega328p microcontroller. With the proof of concept working on the STM32 platform, the next step is porting the project to a smaller controller. As I only have a ATMega328p at my disposal, I chose this microcontroller for the next step in the project. Again, **THIS IS NOT A MEDICAL DEVICE, I AM NOT A MEDICAL PROFESSIONAL.**

1 Introduction

With the motivation for this project documented in the proof of concept (`https://github.com/nt-com/CTmeasure`), I will keep the introduction short and only describe the changes made in this most recent design iteration.

The changes made are the following:

1. As described in the proof of concept, the project now features a 16-bit timer to trigger temperature measurements
2. I programmed baremetal software for the ATMega328p to reduce the code size. However, I am using the supplied AVR headers
3. Because concerns about the readability of the temperature output due to peaks in the measurement, I modified the UI. I am still not satisfied with the UI, it does what it should do but it is still a hacked together mess.

2 Software Microcontroller

2.1 Software Modules Microcontroller

The software consists of three main parts:

1. The drivers (UART, I2C, 16-bit Timer)
2. The interfacing to the MLX90614 sensor
3. The application itself carrying out the timed measurements

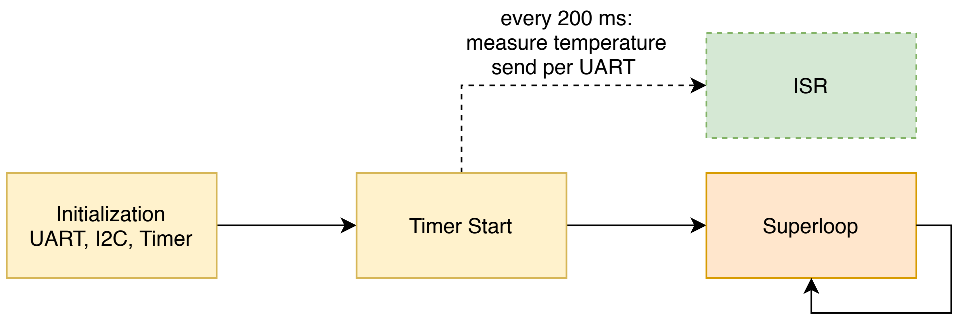
The inline documentation of the code is extensive and references all parts of the ATMega328p datasheet needed to understand it. Further, relevant formula are explained in the inline documentation. An overview of the interfaces is given in the header files, details explained in the c files. The files included are:

- twi.h twi.c (I2C interfacing)
- uart.h uart.c uart_cfg.h (UART interfacing)
- tim16.h tim16.c tim16_cfg.h (16-bit timer)
- debug.h debug.c debug_cfg.h (debugging helpers)
- mlx90614.c mlx90614.h (sensor interfacing)
- main.c

2.2 Program Flow

The software does an initialization of all the microcontroller peripherals in the main.c file in a first step. This includes the UART, I2C and the 16-bit timer. After that the software enters the superloop (while(1)). The measurements and the subsequent data transmission to the PC are done in an interrupt service routine (ISR). This ISR is triggered every 200ms. That means five measurements are done per second and the results are transmitted to the PC. This is depicted in fig. 1.

Figure 1: Flow of the Microcontroller Program.



A major chance is that instead of doing the expensive floating point calculations on the microcontroller, only the raw sensor data is sent to the PC. Here, the UI must do the necessary calculations to convert the raw sensor values to a human readable temperature.

2.3 Debugging the Microcontroller Code

In order to be able to debug the microcontroller code, I added debugging functionality. There are two main parts to the debugging module. First, debugging is turned on an off in the file debug_cfg.h. When debugging is on an LED connected to pin D6 of the microcontroller is initialized. This LED is off until the timer ISR starts, where it is toggled each time the ISR is executed. Further, error codes, if there are any errors, are transmitted through the UART. See the inline documentation for details. What I will mention here is that information about the initialization routines is sent when debugging is activated, as well as information about each step of the I2C process (if there are errors).

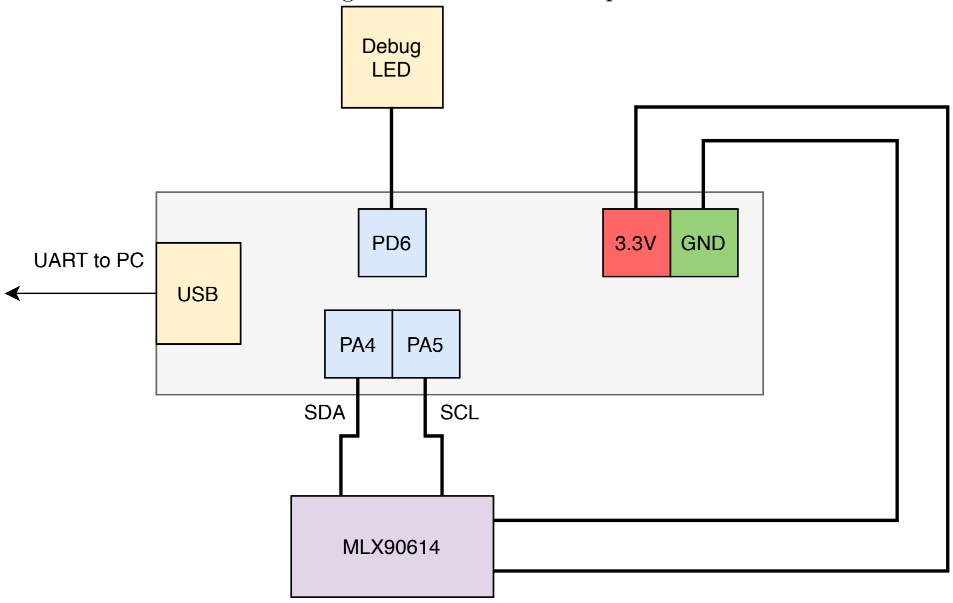
I use preprocessor flags to activate/deactivate the debugging. This makes the code a little hard to read, but is worth a lot when problems are encountered. One note here: The routine to interface with the MLX90614 always returns a status code. When debugging is off, this return value is simply ignored in the software in the current state.

3 Hardware

As a test platform I am using a cheap Arduino Nano clone, I have nothing else at my disposal. This has consequences for the project. First, I can't add the UART bluetooth module as I did in the proof of concept, the UART is already used on this board. I know that there is an Arduino software serial library to get around this, but I don't want to use a bit-banged UART (concerns about stability and speeds). Thus, the bluetooth module is gone, instead I am connecting per cable to the Nano to get the temperature values per serial interface.

As far as I2C is concerned, no external pullups are needed. The sensor features $10k\Omega$ pullups. The sensor is powered via the board's $3.3V$ output. The only other (not strictly necessary component) is the debug LED that does not need to be included and is deactivated once debugging mode is turned off. The complete setup is shown in fig. 2. If a custom board would be made, the UART could be used for a bluetooth module as in the proof of concept. The only change in the software would be, if necessary, a different UART baudrate.

Figure 2: Hardware Setup.



4 User Interface

The user interface is still a hacked together mess and I am confident it could be done much better by someone with more experience in this area. However, it does it's job and that is to show the temperature currently being measured. In this new design iteration of the project, there are two interfacing scripts, both written in python.

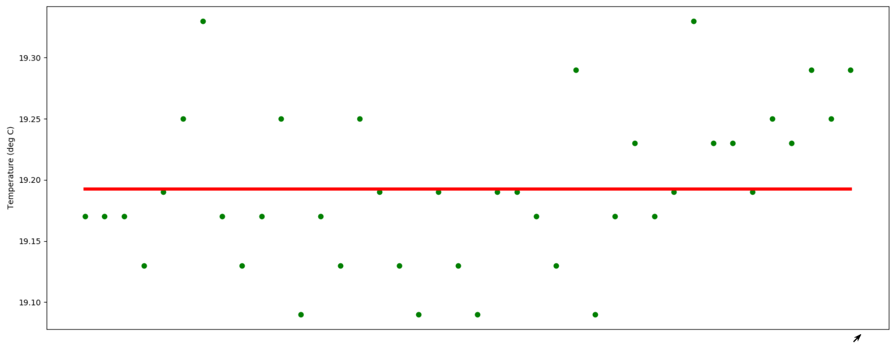
One script shows the sensor values as floating point numbers when started and has no graphical output. The graphical output script is pretty much the same as in the proof of concept, with two modifications:

1. The raw temperature value must be converted to a floating point value, as with the script showing the raw values.
2. During the time temperature is measured, the mean value of all temperatures currently in the measurement window is calculated and displayed.

The second point gets rid of any peaks and bumps and so on in a simple manner, providing the user of the project with a more stable temperature output. However, this only a quick, simple fix one can do. Maybe there is a better way to get a smooth output.

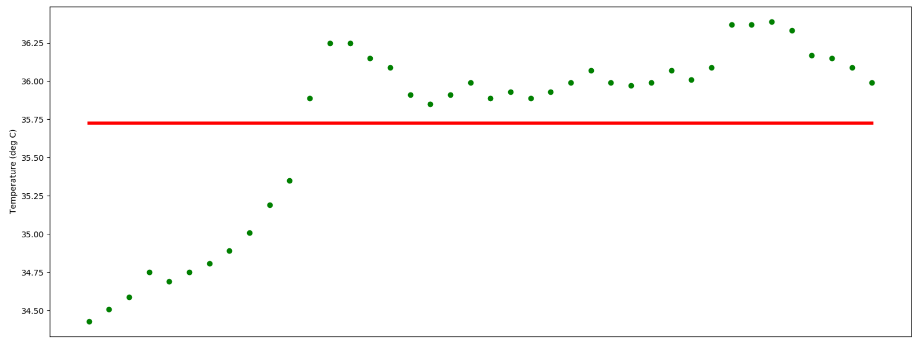
An example of this is shown in fig. 3 showing an ambient temperature measurement. The green dots are samples taken in $200ms$ intervals, the red line shows the average temperature over the measurement window. I can confirm the accuracy through an analog thermometer (more or less as it is analog).

Figure 3: Ambient Temperature Measurement.



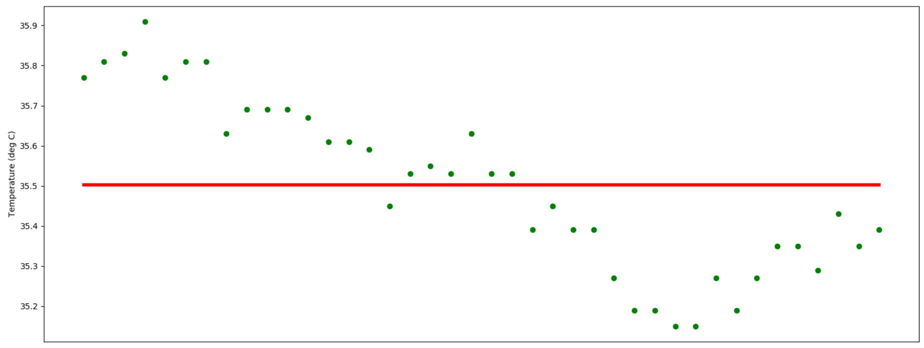
The next measurement is again a forehead measurement, depicted in fig. 4. Here, I measured for approximately 30 seconds, with a distance of about 1-2 centimeters to the sensor.

Figure 4: Forehead Temperature Measurement 1.



In fig. 5 you can see that the measurement stabilizes after about $30 - 40$ seconds.

Figure 5: Forehead Temperature Measurement 2.



5 Conclusion

The software got simpler and smaller, the user interface is a little more readable. However, there is still more work to be done. I am confident the microcontroller application can function on a smaller microcontroller still. All in all, the current state of the project provides a stable base for future developments.

6 Future Work

First step is to get a nicer user interface, the second step is to further modify the microcontroller software for more efficiency. Another topic of interest is power consumption, especially for a battery powered version of the project in later design iterations. Further, a display connected directly to the microcontroller showing the measured temperatures may be an alternative to using a PC for showing the sensor data.