

## Temperature Sensor with Fan Control

### Abstract

The overall motivation for this project was to learn how to use the ADC converter on the TS-7250 board. I had never used this before and wanted to implement a temperature sensor that could be used to control a fan. My program will detect when an overheating condition occurs and turn on the speaker to represent the fan being turned on. When the temperature drops back below the specified temperature, the fan will turn back off again.

### Introduction

My program consists of two processes: the kernel space process and the userspace process. I will utilize the kernel space to detect button presses in real time. The userspace process reads the ADC and displays the user options. The objective of this project was to learn to use analog to digital conversion and to handle overheating conditions by turning on and off a fan. This required communications between the userspace and kernel processes.

### Background

In order to get temperature readings, I purchased a TMP36 analog temperature sensor. I ran this off the 5 VDC power that comes off the LCD pins on the TS-7250. In order to get the sensor to output voltages proportional to 1mV/degree Fahrenheit, I read the data sheet of the TMP36 and followed their suggested schematic shown in Figure 1 below. Once I got the circuit outputting correct voltages, I could begin getting the ADC configured in my program.

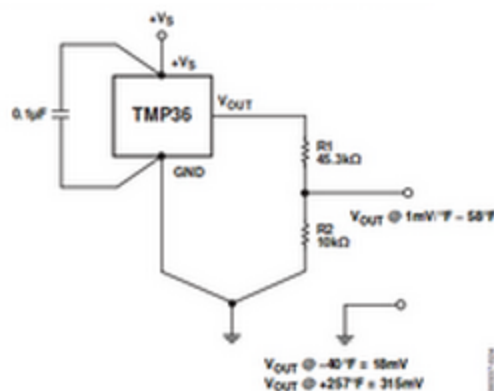


Figure 1.

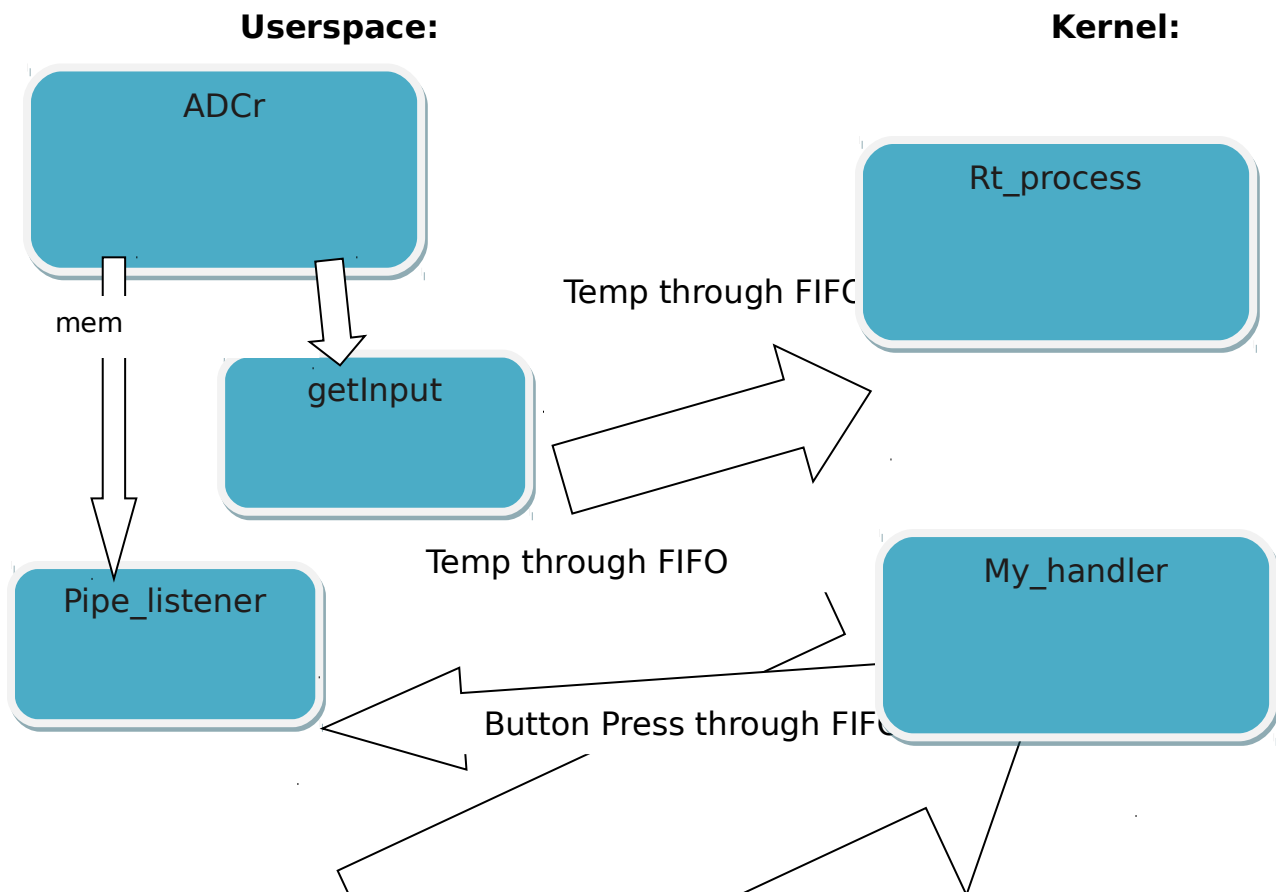
## Implementation

The first program that I will discuss is the kernel module that consists of a GPIOB hardware interrupt handler to detect button presses. This is used to get instantaneous temperature readings. Inside this kernel I run two real time tasks. The first real time task is called **rt\_process** and it checks if the temperature is above 75 degrees Fahrenheit and if it is, it will turn on the red light and sound the speaker to indicate that there is an overheat condition. If the temperature is equal to or below 75 degrees it will turn the green light on to indicate a non-overheating condition.

The ADC uses a 12 bit (0 - 4096) conversion. I set up the ADC to run on a 0- 5 VDC range. My TMP36 is set up to vary by only 1mV/degree Fahrenheit so therefore I am not utilizing my 5 VDC range very effectively. This causes a lot of fluctuation in my temperature readings. The second real time task is called **rt\_process2** and it reads the temperature from the fifo and will use a global variable to let **rt\_process** one update the fan/light accordingly. (I removed **rt\_process2** and read in **rt\_process** now explained why in Results section). The third process inside the kernel is the GPIOB hardware interrupt handler (**my\_handler**). Inside here, I disable interrupt handling, send to the fifo that the button was pressed, clear the interrupt, and re-enable interrupt handling. Inside the **init\_module**, I configure all the GPIOB registers for handling the interrupt for falling edge sensitive. I also enable the debounce register. I also configure the PBDDR, PBDR, PFDDR, PFDR registers for the lights and fan. Finally, I can start the real time timer, set the period and initialize my real time task. I chose a period of 1.2mS for the speaker because this makes a "nice" sound. I also create my real time fifos in the **init\_module**. Lastly, my **cleanup\_module** takes care of deleting the real time task, stopping the timer, releasing the interrupt, and destroying the fifos.

In my userspace program, I spawn 3 pthreads. One is used as a pipe listener because read is a blocking function in userspace. This process simply loops and waits for the kernel to write to the fifo that the button was pressed. Next, this **pipe\_listener** will grab the semaphore and write back to the kernel the current temperature. This happens when it reads that a button was pressed from the FIFO. The second pthread is used to get **user input**. It displays the menu and gives options for checking the current temperature, the maximum temperature, the minimum temperature and exit. If the user selects option 1(current temperature) the program will write to the fifo what the current temperature is. It must also grab the semaphore before it writes to ensure that both pthreads do not try and write at the same time. The third pthread is called **ADCr** and its job is to read the temperature from the MAX197 A/D header. In order to initialize this, I needed to configure the register at 0x10F00000. I set mine to be 0x40 for channel 0, 5V range, unipolar. Bit 7 on register 0x10800000 will go low when the conversion is complete. I then used a short variable to read the value at location 0x10F00000. This is the value between 0 - 4096 that corresponds to the temperature. I simply performed a DC offset of 20 to this to get the temperature value. In main all that is needed is to open the fifo, create pthreads and join the pthreads.

FlowChart:



## Experiments and Results

I know that `rtf_get` was blocking so I was able to put that in the same `rt_process`. At first I was thinking this would need its own process but when I realized that it was non-blocking I moved it to where I would read at the end of every loop iteration. I tested my program by running both the kernel and the userspace at the same time. I would try each option on the menu and at first I was noticing that if the button was pressed and a current temperature reading was made, the `rt_process` was not receiving the temperature. In order to fix this, I added a write to the fifo in the `pipe_listener` after it detects a button press. Sample output is shown below in Figure 2. As you can see from these results, The temperature readings vary between 125 and 48 degrees. This seems like a large variance but when we consider that the 0-5V range is going from 0 to 4094 and the ADC is only outputting a voltage of 1mV/degree it makes sense. In order to decrease this, I would need to amplify the TMP36's output to vary by

100mV/degree to make it more stable. How it is implemented now, 1 mV noise would cause the reading to be off by almost 1 degree.

After running my program, the results indicated that the ADC was reading correct data, just data that was filled with noise. However, the red light and speaker would turn on whenever the temperature was above 75 degrees and the green light would turn on when the temperature was equal to or below 75 degrees. I tested this at least 50 times by either selecting option 1 on the menu or pressing the button. I noticed that since I used a real time task with an interrupt on port B that when I press the button really fast it still works properly. It will turn the lights and/or fan on/off without any noticeable delay.

```
2
The maximum temperature is: 125
Please enter the number of which you would like: (or Push Button for current temp)
1. Check current temperature.
2. Check max temperature.
3. Check min temperature.
4. End.
3
The minimum temperature is: 48
Please enter the number of which you would like: (or Push Button for current temp)
1. Check current temperature.
2. Check max temperature.
3. Check min temperature.
4. End.
1
The current temperature is: 60
Sending temp 60 through fifo.
Please enter the number of which you would like: (or Push Button for current temp)
1. Check current temperature.
2. Check max temperature.
3. Check min temperature.
4. End.
Button was pressed: temp is: 63
Sending temp 63 through fifo.
Button was pressed: temp is: 63
Sending temp 68 through fifo.
Button was pressed: temp is: 68
Sending temp 68 through fifo.
Button was pressed: temp is: 68
Sending temp 68 through fifo.
Button was pressed: temp is: 68
Sending temp 68 through fifo.
Button was pressed: temp is: 68
Sending temp 68 through fifo.
Button was pressed: temp is: 122
Sending temp 122 through fifo.
Button was pressed: temp is: 122
Sending temp 122 through fifo.
1
The current temperature is: 126
Sending temp 126 through fifo.
Please enter the number of which you would like: (or Push Button for current temp)
1. Check current temperature.
2. Check max temperature.
3. Check min temperature.
4. End.
Button was pressed: temp is: 74
Sending temp 74 through fifo.
Button was pressed: temp is: 74
Sending temp 74 through fifo.
```

Figure 2.

## Conclusions

As with any project, there were multiple issues I faced when trying to implement this project. The first major one was not getting the ADC to read any values. After looking this over for a while and testing my TMP36's output using a multi meter and power supply in the Capstone lab, I was able to find out that there was a faulty connector that I was using to power my circuit. Once this was resolved and I was actually getting output on the TMP36 I was able to get readings on the ADC. I also faced a confusing issue with the TS-7250 manual. It states that both 0x1080 0000 and 0x2240 0000 can be polled to determine when the conversion is complete. After looking at sample code from Luis I decided to use the 0x1080 0000 register to determine when it was complete.

Like I previously mentioned, at first glance it may seem that the output from the ADC was incorrect because it was varying so much. However, only 40mV of noise could cause the temperature to vary by 40 degrees. I tried to cut down on the noise by adding a .1uF capacitor as it shows in the TMP36 datasheet and didn't notice that this helped at all. Overall, I learned how to communicate between the kernel and userspace program using FIFOs. I learned that in the kernel reads are non-blocking so you don't need a special process to wait for input. However, in userspace you need a special thread to read because it is blocking. Most importantly, I learned how to set up the analog to digital converter using the TS-7250. It was very rewarding to learn how to get hardware and software to work together. There is virtually no other class where I've gotten to do this kind of project—therefore, I learned valuable skills that no other class has taught me. If I had more time/money I could have tried to amplify the signal of the TMP36's output so that I could more efficiently use the whole 5V range on the ADC.