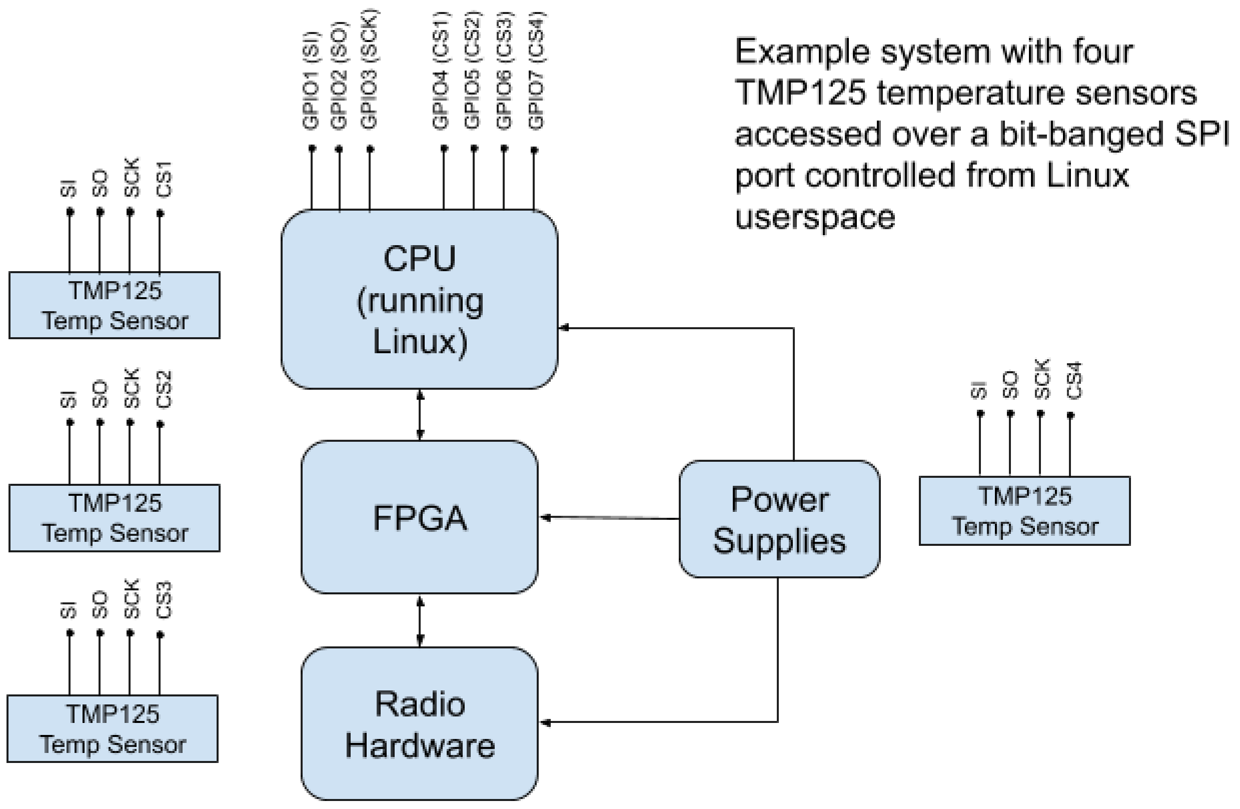
Epiq Solutions' Coding Challenge - December 2019

Development of a Platform Temperature Manager

Overview

Epiq develops a number of different software defined radio platforms consisting of RF front ends, FPGAs, and Linux-based computers hosting all of the processing for the platform. All of the aforementioned electronics generate heat, and thus it is often desirable to monitor the temperature of various points on the radio platform to ensure nothing gets too hot (or too cold in some cases). This is typically accomplished by including temperature sensor chips placed near different components known to generate heat (such as the FPGA, the radio hardware, or the CPU). A typical block diagram of a platform with four temperature sensors is shown below, and this block diagram will be used for this coding project. The TMP125 temp sensor from Texas Instruments is used for this particular system, whose datasheet can be found at the following link:

https://www.ti.com/lit/ds/symlink/tmp125.pdf



Challenge Part 1:

In C or C++, develop a Platform Temperature Manager (PTM) software application that is responsible for monitoring the temperature of the system and reporting any over/under temperature conditions. This application should have the following characteristics:

- Runs in a Linux environment (we typically use Ubuntu as our development environment, so that is the preferred distribution for this exercise)
- Automatically checks the temperature of each of the four sensors once every 500 mS
- Reports every over-temperature (above +85 degrees C) or under-temperature (below -40 degrees C) as an "alarm condition" by printing a relevant alarm message to stdout containing the current date and time, the temp sensor that alarmed, and the temperature measurement

For Part 1 of this challenge, assume there is already an existing API that can be used by the PTM for measuring the temperature of a given temp sensor (we are using Texas Instruments' TMP125 temp sensor for this project), with the following signature:

```
/* Description: Read the temperature of the requested temperature sensor_id
   Parameters: temp_sensor_id-the id of the temp sensor to read (1-4)
               p_temp_in_degrees_c-a pointer to a float where the temperature
               will be written
   Returns: int32_t indicating status (0=success, any other value indicates an
            error code)
*/
int32_t tmp125_read_temp(uint8_t temp_sensor_id, float* p_temp_in_degrees_c);
```

(you may want to mock up this function to return known or random temperature readings for the purpose of testing out your PTM implementation)

Deliverables

The final delivered project should include the build instructions (in whatever form you prefer) and source code for the PTM written in C or C++. We should be able to build and test out your PTM application on a standard Ubuntu system.

Bonus #1:
Make the PTM a Linux library instead of a stand-alone application, which can then run in its own thread as part of a main host application. This would allow multiple applications to include this library to perform temperature monitoring as part of its core functionality.

Bonus #2: Provide an API to allow the main application utilizing the PTM library to asynchronously check the temperature of any of the four sensors at any time. The signature of this API accessible to the main application should be defined as follows:

```
/* Description: Read the temperature of the requested temp sensor
   Parameters: temp_sensor_id-the number of the sensor to read (1-4)
               p_temp_in_degrees_c-a pointer to a float where the temperature shall
               be written in degrees celsius
   Returns: int32_t indicating status (0=success, anything else indicates an
            error code)
*/
int32_t PTM_read_temp(uint8_t temp_sensor_id, float* p_temp_in_degrees_c);
```

Bonus #3: Provide a creative solution for publishing alarm messages. This could be by sending an alarm message over a socket to a TCP server, posting a notification to a webpage, sending an email, or anything else you'd deem as an interesting way to send this alarm. Using existing code libraries is perfectly acceptable here, so long as the original source is referenced.

Challenge Part #2

We often have to write software drivers for different hardware peripherals, including things like temperature sensors such as Texas Instruments' TMP125 described here:

https://www.ti.com/lit/ds/symlink/tmp125.pdf

In Part #1 of this challenge, we could assume that there was a driver available to provide temperature readings (such as tmp125_read_temp() function). In this section, you'll be writing the driver for the TMP125 temp sensor itself. This software driver can execute in Linux userspace (i.e., it doesn't need to be a proper Linux kernel space driver). As described in the above datasheet, the TMP125 is accessed through a SPI interface. For this challenge, you'll be bit-banging the SPI master interface using a simple set of hardware access functions that can be assumed to already exist on the host Linux system for accessing GPIO pins from userspace. The code for these functions are provided at the end of this challenge for reference in the gpio_lib.c file.

Deliverables

The final delivered driver should implement a Linux userspace software library in C or C++ for interacting with the TMP125 temperature sensor over a bit-banged SPI interface. Note that there are a total of four TMP125 sensors in the system, and the driver must be able to access all four of them. This library shall implement the following functions:

```
/* Description: Provide any necessary initialization of the library
   Parameters: none
   Returns: int32_t indicating status of the init operation (0=success, anything else indicates an
            error code)
 */
int32_t tmp125_init(void);

/* Description: Read the temperature of the requested temp sensor in degrees C
   Parameters: temp_sensor_id-the id of the temp sensor to read (from 1 to 4)
               p_temp_in_degrees_c-a pointer to a float where the temperature will be written
   Returns: int32_t indicating status (0=success, any other value indicates an error code)
*/
int32_t tmp125_read_temp(uint8_t temp_sensor_id, float* p_temp_in_degrees_c);
```

It won't be necessary to test out the TMP125 software driver on a real system (since this would require hooking in to actual hardware, or emulating the SPI slave capabilities of the TMP125 temp sensor), so for this coding challenge, please just be sure that the library compiles cleanly. The source code for a simple test application demonstrating the functionality of the TMP125 driver shall also be included, as well as the source code for the TMP125 driver and any needed build instructions.

```
/*      File: gpio_lib.c
        Description: Provides a mock up of an API to control GPIO pins from Linux userspace for
        the TPM coding challenge
        Author: John Doe, Epiq Solutions
*/

/* Includes */
#include <stdio.h>
#include <stdint.h>
#include “gpio_lib.h”

/* Local Defines */
#define NUM_PORTS 1 /* we have a single 8-bit GPIO port available */
#define NUM_PINS_PER_PORT 8  /* each GPIO port has 8-bits, represented as 0-7 */
#define DIR_OUTPUT 0         /* set dir to 0 for output */
#define DIR_INPUT  1         /* set dir to 1 for input */

/* Local Variables */
/* For this mock up library, the pin values and direction are stored in simple arrays static to this
file */
static uint8_t gpio_port_pins[NUM_PORTS];
static uint8_t gpio_port_dirs[NUM_PORTS];

/* Global Functions */

/* Description: set the GPIO direction for the specified GPIO pin pin_id on GPIO port port_id.
Parameters: dir=0 indicates output, dir=1 indicates input,
Returns: int32_t indicating status (0=success, otherwise an error code) */
int32_t gpio_set_direction(uint8_t port_id, uint8_t pin_id, uint8_t dir)
{
        int32_t status=0;
        if (port_id >= NUM_PORTS)
        {
                printf(“Error: invalid port_id %d;\r\n”,port_id);
                status = -1;
                goto out;
        }
        if (pin_id >= NUM_PINS_PER_PORT)
        {
                printf(“Error: invalid pin_id %d; valid pin_id values are between 0 and %d\r\n”,pin_id,
                       (NUM_PINS_PER_PORT-1));
                status = -2;
                goto out;
        }

        if (dir == DIR_OUTPUT)
        {
                gpio_port_dirs[port_id] &= ~(1<<pin_id);
        }
        else if (dir == DIR_INPUT)
        {
                gpio_port_dirs[port_id] |= (1<<pin_id);
        }
        else
        {
                printf(“Error: invalid dir %d (0=output, 1=input)\r\n”,dir);
                        status = -3;
```

```
            goto out;
        }

        printf("Info: successfully set pin %d of port %d to a direction of %d\r\n", pin_id, port_id,
               dir);

out:
        return(status);
}

/* Description: reads the pin value for the specified GPIO pin pin_id on GPIO port port_id.
   Parameters: port_id is the ID of the GPIO port, pin_id is a value from 0-7, and pin_state=0 for a
               logic level low, pin_state=1 for logic level high
   Returns: int32_t indicating status (0=success, otherwise an error code)
  */
int32_t gpio_read_pin(uint8_t port_id, uint8_t pin_id, uint8_t* p_pin_state)
{
        int32_t status=0;

        if (port_id >= NUM_PORTS)
        {
                printf("Error: invalid port_id %d\r\n",port_id);
                status = -1;
                goto out;
        }
        if (pin_id >= NUM_PINS_PER_PORT)
        {
                printf("Error: invalid pin_id %d; valid pin_id values are between 0 and %d\r\n",
                        pin_id, (NUM_PINS_PER_PORT-1));
                status = -2;
                goto out;
        }

        if ((gpio_port_pins[port_id] & (1<<pin_id)))
        {
                *p_pin_state = 1;
        }
        else
        {
                *p_pin_state = 0;
        }
        printf("Info: successfully read pin %d of port %d with a value of %d\r\n", pin_id, port_id,
               *p_pin_state);

out:
        return(status);
}


/* Description: writes the pin value for the specified GPIO pin pin_id on GPIO port port_id.
   Parameters: port_id is the id of the GPIO port, pin_id is a value from 0-7, and pin_state=0 for a
               logic level low, pin_state=1 for logic level high
   Returns: int32_t indicating status (0=success, otherwise an error code)
  */
int32_t gpio_write_pin(uint8_t port_id, uint8_t pin_id, uint8_t pin_state)
{
        int32_t status=0;
        if (port_id >= NUM_PORTS)
        {
                printf("Error: invalid port_id %d\r\n",port_id);
                status = -1;
            goto out;
        }
        if (pin_id >= NUM_PINS_PER_PORT)
        {
                printf("Error: invalid pin_id %d; valid pin_id values are between 0 and %d\r\n",
                        pin_id, (NUM_PINS_PER_PORT-1));
                status = -2;
                goto out;
        }

        if (pin_state == 0)
        {
                gpio_port_pins[port_id] &= ~(1<<pin_id);
        }
        else if (pin_state == 1)
        {
                gpio_port_pins[port_id] |= (1<<pin_id);
        }
        else
        {
                printf("Error: invalid pin state %d requested\r\n", pin_state);
                status = -3;
                goto out;
        }

        printf("Info: successfully wrote pin %d of port %d with a value of %d\r\n", pin_id, port_id,
               pin_state);

out:
        return(status);
}
```