# Exercise in Embedded Computing: I$^2$C

Sivan Toledo, Tel-Aviv University

November 14, 2009

The purpose of this exercise is develop a driver for the I$^2$C peripheral. The goal is to write a program that reads the temperature from the LM75 temperature sensor about once per second and prints it to a serial connection (UART0).

1. Download the UART driver `uart0.c`, the print formatter `print.c`, and the test program `test-uart.c` from `exercises/uart` and build and test the code. To see the output, add the flag `-term` to the `program` target in the makefile (that is, run `lpc21isp` with this flag).

2. Implement a master-mode I$^2$C driver `i2c.c`. The driver should be interrupt driven. Its interface should include the following functions.

   ```
   void i2cInit();
   ```

   This function should initialize the peripheral. It should set the bit frequecy to the value given by the constant `I2C_BIT_RATE` which must be defined before `i2c.c` is included in the application program. To set the bit frequency, the function needs to know the frequency of the clock that feeds the I$^2$C clock generator; this frequency is given by the constant `CLOCKS_PCLK`.

   ```
   int32_t i2cMasterTransact(uint8_t  slave_address,
                             uint8_t* command,
                             int32_t  command_len,
                             uint8_t* response,
                             int32_t  response_len);
   ```

   This function writes the command to a slave with a given slave address and reads a response from it. If the command has zero length, the function only reads from the slave. If the length of the response buffer is zero, the function does not read from the slave at all. The function returns the number of bytes that it read from the slave (if the slave responds with a NACK before sending the expected number of bytes, the master should stop the transaction). If the transaction fails (for example, if the slave does not acknoledge its address), the function returns a negative number that indicates the nature of the failure.
   Even though the driver is interrupt driven, this function is blocking. It starts the I$^2$C transaction and then waits in a while loop for the ISR to signal that

the transaction is complete or was aborted due to an error (e.g., the master lost arbitration, or the slave does not respond to its address, etc.). It should detect excessive waiting and return with an error code that indicates a time-out. Timeouts can occur if the bus is not ready to begin the transaction and if the clock is stretched for two long. The maximum time for a transaction should be one second, to allow for even very slow slaves.

3. Test the driver by trying to read two bytes from the LM75, whose address is `0x90`. That is, the address byte should contain `0x90` when we write to the slave (not the case for this part of the exercise) and `0x91` when we read from it. (The documentation of $I^2C$ slaves sometimes specifies addresses this way and sometimes as a 7-bit number which is stored in the 7 most significant bits of the first byte sent by the master.) The LM75 can support clock rates of up to 400 kHz. Here is an example of what the relevant parts of the code should look like:

```
#define CLOCKS_PCLK 3000000
#define I2C_BIT_RATE 400000
#include "i2c.c"
...
  return_code = i2cMasterTransact(0x90,0,0,response,2);
```

4. The program should print the temperature in degrees Celsius. Read the data sheet of the LM75 in order to interpret the format of the two bytes that it returns.

5. How many LM75's can serve as slaves on one $I^2C$ bus?

6. What happens if you clock the LM75 too fast? Up to what actual bit rate can you does it work?

7. Additional ideas:

   (a) Display the temperature on the LCD.

   (b) The $I^2C$ bus on the LPC2148 Education Board gives you access to another chip, a CAT1025, which contains 256 bytes of non-volatile memory. It also runs at up to 400 kHz. Write a driver for the CAT1025; it should use the $I^2C$ driver to access the chip but provide higher-level functions.

   (c) Design and implement a non-blocking interface for the $I^2C$ driver, so that `i2cMasterTransact` does not wait for the transaction to end. Instead, it should merely start the transaction and return. When the transaction ends, the ISR should call a completion handler. Convert your program to use this driver.