Desenvolvimento de um Oxímetro de Pulso Usando Microcontrolador MSP430

Davi de Alencar Mendes, João Paulo Sanches Guimarães

***Resumo*—No presente trabalho, propomos e demonstramos o desenvolvimento de um Oxímetro de Pulso usando o Microcontrolador MSP430FR2433 da *Texas Instruments*. A Saturação de Oxigênio SpO_2 e a Frequência Cardíaca são parâmetros chave para o monitoramento da saúde de pacientes. O sistema proposto consiste em um sensor de SpO_2, MSP430FR2433 e um *display*. A saturação do oxigênio é calculada a partir da razão entre duas intensidades de luz, já a frequência cardíaca é calculada a partir da diferença de tempo entre dois picos da intensidade do sinal infravermelho. Os parâmetros medidos são processados no microcontrolador e exibidos no *display*.**

***Index Terms*—Oxímetro de Pulso, SpO_2, Frequência Cardíaca, MSP430.**

I. Introdução

NAS últimas décadas observa-se uma crescente preocupação com assuntos relacionados a saúde. Em um contexto normal de monitoramento da saúde de pacientes, tem-se grandes restrições em mobilidade e usabilidade de tal maneira que soluções portáteis se tornam necessárias para diversos tipos de pacientes. O gás oxigênio é parte integrante dos processos biológicos que ocorrem no corpo humano. O transporte desse importante gás ocorre através das hemoglobinas nas células vermelhas do sangue. Informações críticas podem ser adquiridas por meio da medição da quantidade de oxigênio presente no sangue na forma de um índice percentual do total da capacidade máxima. O oxímetro de pulso é um instrumento que realiza tal medida [1].

O oxímetro de pulso inclui dois diodos emissores de luz (*LEDs*), um no espectro vermelho visível (660nm) e outro com espectro infravermelho (940nm) [2]. Mudanças na intensidade da luz transmitida pelos tecidos causadas pela pressão arterial sanguínea são detectadas como um sinal de voltagem pelo fotopletismógrafo (sensor SpO_2). No oxímetro apresentado será utilizado um sensor que adota o método de reflectância em sua operação, ou seja, há um emissor de luz ao lado de um fotodetector que mede a resposta após a emissão de luz. A Figura 1 mostra que há uma absorção constante de luz sempre presente devido aos diferentes tecidos presentes, sangue venoso e sangue arterial.

Davi de Alencar Mendes é estudante de Graduação em Engenharia Eletrônica pela Universidade de Brasília - UnB, Brasília, Brasil. Email institucional: dmendes@aluno.unb.br - Matrícula: 16/0026415

João Paulo Sanches Guimarães é estudante de Graduação em Engenharia Eletrônica pela Universidade de Brasília - UnB, Brasília, Brasil. Email institucional: sanches.joao@aluno.unb.br - Matrícula: 16/0031923

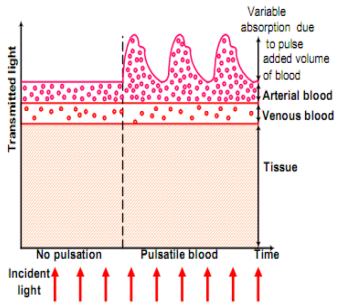

Figura 1. Luz transmitida em fluxo sanguíneo no tempo

Entretanto, a cada batimento cardíaco há um deslocamento de sangue arterial que provoca o aumento do volume de sangue que transita pelo espaço de medição do oxímetro, causando maior absorção de luz durante esse período caraterístico [1]. Se o sinal adquirido pelo fotodetector for analisado como um sinal de onda é possível observar que há picos a cada batimento e vales entre os batimentos. Com a luz absorvida em um vale (sinal que deve incluir todas as absorções constantes) for subtraída de uma amostra de pico é obtido o resultado do volume de sangue arterial trazido a cada batimento. Com esses sinais constantes e sua variação calcula-se um valor intermediário chamado R, a Razão Normalizada. Usando R, podemos calcular SpO2 usando a fórmula [1]:

$$SpO2 = 110 - 25 * R$$

O MSP430 incorpora uma CPU RISC de 16-bits, periféricos e um sistema de *clock* flexível que se interconecta usando uma arquitetura Von-Neumann com barramento comum de memória e dados. Com suporte para periféricos digitais e analógicos, o MSP430 oferece soluções para aplicações que usam diferentes sinais.

II. Desenvolvimento

A. Panorama do Protótipo Funcional

Para o ponto de controle 3 é esperado que seja obtido um protótipo funcional que atenda os principais requisitos do projeto. Para o Oxímetro de pulso é esperado que os seguintes requisitos funcionais sejam desenvolvidos:

- [RF01] O sistema deverá exibir em um *display* os dados obtidos.
- [RF02] O sistema deverá alertar o usuário com avisos sonoros.
- [RF03] O sistema deverá processar os dados obtidos do sensor.
- [RF04] O sistema deverá se comunicar com o sensor utilizando o protocolo I^2C (*Inter-Integrated Circuit*).

Visando alcançar estes requisitos, foi utilizada a plataforma de desenvolvimento *Code Composer* da *Texas Instruments*. Esse ambiente integrado de desenvolvimento permite desenvolver aplicações em C e C++ para o MSP430 com uma grande variedade de recursos adicionais como controle de versão e depuração de código.

Como forma de Revisão bibliográfica foram pesquisadas bibliotecas de Arduino [3],[4] que implementam a comunicação e o processamento dos dados Obtidos do sensor MAX30100. O *datasheet* do sensor[2] foi utilizado para guiar o desenvolvimento das funções que são utilizadas para controlar os estados de operação e transferência de dados do sensor. O sensor utiliza uma estrutura de dados em Fila (*FIFO - First In First Out*) para armazenar os valores amostrados em 16 *bits* das leituras já realizadas do fotodiodo para o espectro Vermelho e Infravermelho. Como consequência, foi necessário implementar um buffer circular[8] para armazenar os dados lidos do sensor.

Para o processamento dos dados coletados do sensor MAX30100 é necessário implementar um filtro passa-baixa do tipo *Butterworth*[7] de primeira ordem com $\alpha = 0.1$ e frequência de corte - 10Hz. Após ser filtrado, o sinal é utilizado no algoritmo de detecção de batimento para estimar a frequência cardíaca utilizando dados de infravermelho. Para estimar os valores de oximetria é necessário obter somente a faixa AC dos dados de infravermelho e vermelho. Nesse sentido, é necessário implementar um filtro removedor da faixa DC[6] do sinal. Após serem filtrados, os dados são utilizados para estimar os valores de SpO_2.

B. O Protótipo Funcional

Para organizar o desenvolvimento de um protótipo funcional que satisfaça os requisitos propostos para este projeto, este foi divido nos seguintes itens:

1) Descrição de Hardware: Para o seu funcionamento, o protótipo utilizará a seguinte lista de materiais:

- MSP430FR2433;
- Sensor MAX30100;
- *Display* LCD 16x2 - JHD 162A;
- *Buzzer* (buzina);

O microcontrolador MSP430FR2433 processará todos os dados e também integrará os periféricos envolvidos no sistema.

O Sensor MAX30100 é um componente fundamental ao protótipo, uma vez que este é responsável por obter e armazenar os dados obtidos pela fotopletismografia, medição feita a partir dos LEDS vermelho e infravermelho. Além desses dados, o sensor armazena em seus registradores a temperatura, parâmetro de extrema relevância para a calibração do sensor.

O *display* LCD será utilizado para transmitir a informação processada pela MSP para o usuário ou para a equipe médica.

O *buzzer* será utilizado como um alerta em situações de emergência que possam apresentar risco para o paciente, devido à baixa oxigenação sanguínea ou à elevada frequência cardíaca.

O arranjo desses componentes foi esboçado por meio do diagrama de blocos representado na Figura 2.

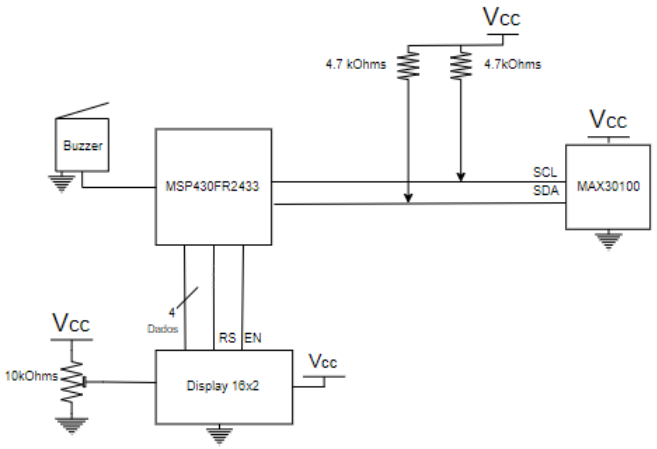

Figura 2. Diagrama de Blocos do Protótipo

Mais detalhadamente, as ligações entre o o MSP e o Sensor MAX30100 foram efetuadas de acordo com o esquemático a seguir:

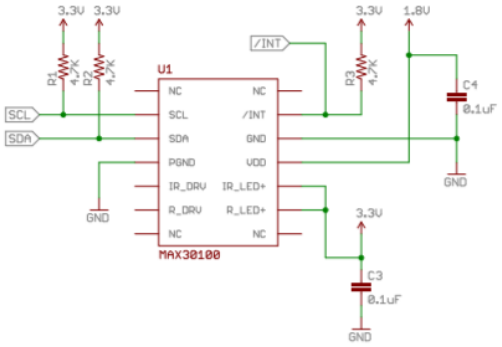

Figura 3. Esquemático de conexões do Sensor MAX3010

Para a versão atual do protótipo funcional vale notar que os pinos: *INT* - interrupção, *IR_DV* e *R_DV* - cátodo dos LEDs não estão sendo utilizados, pois não há necessidade de utilizar nenhuma dessas funções. É interessante notar que foram utilizados resistores de *pull-up* externos nos pinos *SDA* e *SCL* para estabelecer a comunicação I^2C (Inter-Integrated Circuit) com o sensor.

2) Descrição de Software: O desenvolvimento do software para o oxímetro foi realizado em arquivos e funções com o objetivo de modularizar o código, facilitar a manutenção e legibilidade. A divisão da partes do projeto foi realizada de maneira a atender os requisitos levantados. As principais partes e o seu respectivo requisito funcional que compõe a aplicação são:

- [RF04] Comunicação I2C - *I2C.c*
- [RF04] Biblioteca do Sensor MAX30100 - *MAX30100.c*
- [RF03] Filtros - *MAX30100_Filters.c*
- [RF03] Calculadora de SpO_2 - *MAX30100_SPO2Calculator.c*
- [RF03] Algoritmo para Detecção de Frequência Cardíaca - *MAX30100_BeatDetector.c*

- [RF03] Algoritmo para Oximetria de Pulso - *MAX30100_PulseOximeter.c*
- [RF01] e [RF02] Exibição em *Display* e *Buzzer* -

Cada uma das partes do projeto interage entre si para formar o conjunto que compõe o oxímetro de pulso. A interação entre as partes descritas e o seu funcionamento será explorado na seção de resultados.

III. Resultados

A. Comunicação I2C

O protocolo de comunicação I2C é o meio de comunicação entre o sensor MAX30100 e o MSP430. Foram utilizados os resistores de *pullup* nas linhas de clock e dados. O código para o protocolo I2C foi reaproveitado a partir do código de exemplo da *Texas Instruments*[9] para o modelo MSP430FR2433 que está sendo utilizado no projeto. O código permite escrever/ler múltiplos *bytes* em um registrador a partir do endereço de dados e do dispositivo. A implementação do protocolo conta com a utilização de modos de baixo consumo e interrupções para recepção (RX) e envio de dados (TX).

As funções de leitura são utilizadas para receber os dados provenientes das conversões AD do sensor MAX30100 e para ler registradores que indicam o estado de operação do sensor. Já as funções de escrita são utilizadas para configurar os modos de operação do sensor e outras funcionalidades. Em especial, a função de leitura do protocolo I2C permite armazenar os *bytes* lidos em um buffer e permite ler diversas vezes o mesmo endereço. Essa funcionalidade de repetição de leitura (*burst-read*) foi utilizada na implementação da biblioteca do sensor MAX30100.

B. Biblioteca do Sensor MAX30100

A biblioteca do sensor MAX30100 é um conjunto de funções que permitem realizar mudanças nos modo de operação e funcionalidades do sensor. Essas funções utilizam o protocolo I2C para realizar o envio/recepção de dados. Dentre as funcionalidades que a biblioteca do sensor permite realizar, podemos destacar:

- Configurar modo de operação: *setMode()*
- Configurar taxa de amostragem do conversor AD: *setSamplingRate()*
- Configurar corrente dos LEDs: *setLedsCurrent()*
- Realizar leituras do sensor de temperatura: *retrieveTemperature()*, *isTemperatureReady()* e *startTemperatureSampling()*
- Ler dados da fila de amostras R e IR: *readFifoData()*
- Resetar dados da fila de amostras: *resetFifo()*
- Retirar dados do buffer circular: *getRawValues()*
- Desligar/Continuar o sensor: *shutdown()*/*resume()*

Inicialmente, o sensor é configurado no modo de operação (SpO_2 ou Frequência Cardíaca), configura-se o conversor AD em sua resolução (10, 12, 14 ou 16 *bits* por amostra) e frequência de amostragem. Posteriormente, a corrente dos LEDS é configurada e o sensor é iniciado. Com o início da coleta de amostras, o sensor insere os valores amostrados na fila de amostras para serem lidos pelo MSP430. Após os dados serem retirados da fila de amostras do sensor e inseridos no buffer circular de dados é necessário processar as amostras para estimar SpO_2 e a frequência cardíaca.

C. Filtros - Butterworth e Removedor DC

O sensor MAX30100 já incorpora algumas funcionalidades para filtrar ruídos como cancelamento de luz ambiente e um filtro na faixa 50/60 Hz para remoção de ruídos indesejados. Para realizar os cálculos de oximetria é necessário avaliar somente a faixa AC das amostras obtidas do sensor. Nesse sentido, foi implementado um filtro que remove a faixa DC do sinal para o espectro vermelho e infravermelho.

Uma das implementações de um filtro que tenha a função desejada é o filtro de média móvel, no entanto alguma das frequências acima do nível DC podem ser atenuadas ao utilizar essa alternativa. Desenvolver um filtro passa-alta do tipo FIR (*Finite Impulse Response*) acarretaria na mesma situação com o adicional de ser um filtro com alto custo computacional para a implementação no MSP430. A alternativa encontrada foi utilizar um filtro IIR (*Infinite Impulse Response*) simples, similar a um ressonador[6]. Esse filtro garante uma boa resposta de atenuação para as frequências DC e pode ser descrito com a seguinte equação:

$$w(t) = x(t) + \alpha \cdot w(t-1) \tag{1}$$

$$y(t) = w(t) - w(t-1) \tag{2}$$

Onde $y(t)$ é a saída do filtro na equação 2, $x(t)$ é a atual amostra de entrada do filtro na equação 1, $w(t)$ é um valor intermediário do filtro e α é uma constante que atua como fator de escala para o filtro. Caso $\alpha = 1$, todos os valores passam pelo filtro. Para valores próximos de 1 o filtro atenuará os valores da faixa DC. O filtro DC do oxímetro de pulso utiliza o valor de $\alpha = 0.95$.

A detecção da frequência cardíaca utiliza o sinal IR (Infravermelho) após ter sido filtrado pelo removedor DC e para remover as distorções harmônicas desse sinal é necessário utilizar um filtro passa-baixa. Embora o filtro *Butterworth* seja um filtro passa-banda ele pode ser configurado como um filtro passa-baixas. Para implementar esse filtro foi utilizada a plataforma *Filtuino*[7] que gera o código em C++ das constantes do filtro para utilização com o *Arduino*. Após algumas modificações o filtro foi testado e adaptado para utilização no MSP430.

Para configurar o filtro é necessário conhecer a frequência de amostragem para qual o sensor MAX3010 está operando. O sensor permite amostrar com frequências de até 1kHz. As atuais configurações adotadas para o sensor adotam a frequência de amostragem de 100Hz. Dessa maneira, o valor de 100Hz (Fs) foi utilizado como frequência de amostragem para o sensor. Para o valor de frequência de corte foi escolhido o valor de 10Hz (Fc). A partir desses valores foi obtido o valor de $\alpha = \frac{Fs}{Fc} = 0.1$. Vale ressaltar que não é preciso obter um filtro com grande precisão já que o objetivo desse filtro é melhorar a qualidade do sinal IR para a detecção de picos. Essa detecção é parte do algoritmo que estima a frequência cardíaca e será descrita posteriormente.

D. Algoritmo para Oximetria de Pulso e Calculadora de SpO_2

O algoritmo para estimação da oximetria recebe os dados após a filtragem DC e também a frequência cardíaca. O algoritmo tem seu funcionamento baseado em uma máquina de estados com 3 estados. Cada um dos estados tem uma função definida para o funcionamento do algoritmo e utilizam funções para calcular o valor de SpO_2. O diagrama de estados pode ser visualizado na figura 4.

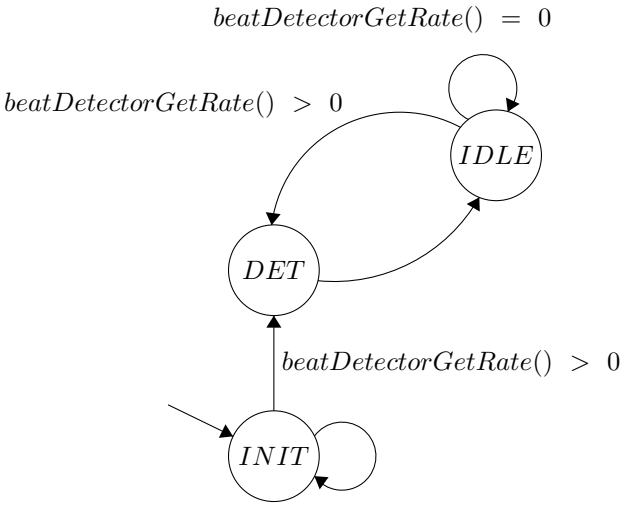

Figura 4. Diagrama de Estados para algoritmo da Oximetria

A máquina de estados inicia no estado *INIT* (início) e somente transita para o estado *DET* quando houver batimento cardíaco detectado - *beatDetectorGetRate() > 0*. O estado de detecção (*DET*) é responsável por atualizar a calculadora de SpO_2, calculando o valor da Razão Normalizada (R) a cada três batimentos detectados. A partir desse valor, calcula-se o valor final de SpO_2. Então, a máquina de estados transita para o estado de repouso (*IDLE*) e se mantém nesse estado até que receba um valor diferente de zero para a frequência cardíaca. Durante o estado de repouso, a calculadora de SpO_2 limpa os valores armazenados até que a máquina de estados retorne ao estado de detecção (*DET*).

E. Algoritmo para Detecção de frequência cardíaca

Para obtenção da frequência cardíaca, utiliza-se uma FSM (máquina de estados finitos) que visa detectar a presença de um sinal biológico estável para efetuar a medição da frequência cardíaca. A FSM tem um estado inicial, *INIT*, onde há um atraso inicial antes da detecção, avançando posteriormente para o estado de espera por um sinal possivelmente válido.

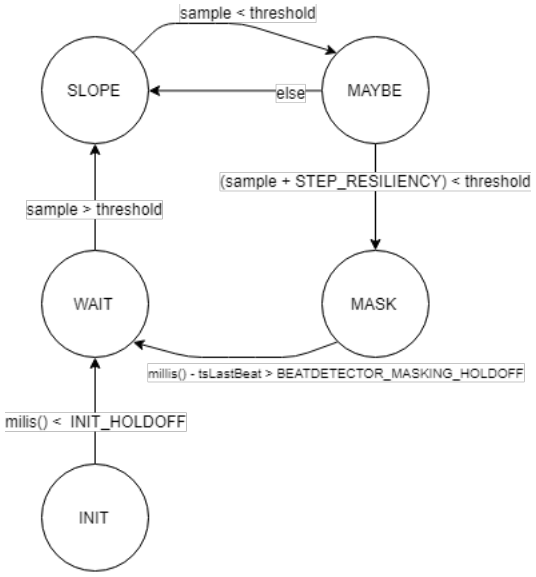

Figura 5. Diagrama de Estados da Frequência cardiaca

Esse estado, denominado *WAIT*, onde é feita a análise da amostra recebida. Caso seja detectada uma amostra maior que o limiar comparativo, este é atualizado e a máquina passa para o próximo estado,. Nesse estado, há o decaimento do limiar a cada ciclo, pois, entende-se que o sinal lido não é mais relevante.

No próximo estado, *SLOPE*, detecta-se um pico, onde compara-se a amostra com o limiar, mudando para o próximo estado quando a amostra for menor que o limiar atual. Caso contrário, atualiza-se o limiar com o mínimo entre a amostra atual e a máxima já lida.

Após o *SLOPE*, há o estado de *MAYBE*, onde considera-se a existência de um sinal de frequência cardíaca estável, após detectar um pico do sinal no estado anterior. Nesse estado, o período entre dois picos é calculado por meio do tempo percorrido entre o último pico e o instante atual. Essa medida é essencial para a estimação da frequência cardíaca.

Se a amostra estiver abaixo do limiar em quantidade maior que um parâmetro definido, entende-se que acabou de acontecer um pico e faz-se a medição descrita acima e o próximo estado será o denominado *MASK*. Caso contrário, espera-se um novo pico, portanto, volta-se ao estado *SLOPE*, visando detectar outro pico.

No estado de *MASK*, há um tempo de espera, onde é analisado o tempo decorrido entre o último pico e o instante atual. Quando este ultrapassa aquele, passa-se para o estado de *WAIT*, reiniciando o ciclo. Enquanto o tempo de espera definido não é ultrapassado, há o decréscimo do limiar.

F. Exibição em Display e Buzzer

Para exibição dos dados com o usuário, é utilizado um *display* LCD 16x2, tanto para a oximetria, tanto para a frequência cardíaca. O *display* se comunica com a MSP430 por meio da emissão de dados paralelos, com o auxílio dos bits RS, EN e RW. É controlável, por meio de configurações enviadas por comandos, o modo de operação do *display*, sendo possível enviar dados em *byte* ou em *nibble*. A comunicação

por *byte* é vantajosa em velocidade, porém utiliza muitos pinos do microcontrolador. Portanto, foi escolhida a comunicação paralela de apenas um *nibble*.

Quanto aos bits de controle, temos que RS identifica o *nibble* enviado ou como dados, ou como comandos, para configuração do *display*; o bit EN (*enable*) é utilizado de maneira a controlar por meio de um pulso o momento no qual os dados são válidos para leitura ou escrita no *display*, enviando ou recebendo o *nibble* de dados; por fim, o bit RW controla o sentido dos dados, variando entre leitura e escrita de acordo com seu nível lógico.

A comunicação é assíncrona, porém utiliza *timers* para auxiliar em atrasos essenciais para o seu funcionamento. Há atrasos necessários de diferentes durações, para envios de dados e comandos, bem como para inicialização do componente.

Um problema encontrado foi a dificuldade da exibição dos dados no *display*. Após análise, concluiu-se que, apesar dos dados de oximetria estarem sendo gerados e processados com sucesso, não foi feita a exibição no *display* com sucesso. Suspeita-se que a taxa de atualização dos dados esteja muito alta e, por isso, o *display* não seja capaz de acompanhar sua atualização, devido aos atrasos necessários citados anteriormente.

O *buzzer*, apesar de ter sua implementação trivial, não foi implementado neste ponto de controle devido ao não funcionamento da obtenção numérica de frequência cardíaca. Uma vez que não se tem este dado, optou-se por adiar a implementação do *buzzer* até o ponto que a frequência cardíaca esteja em pleno funcionamento.

IV. Conclusão

Após a execução do projeto descrito neste documento, foram obtidos resultados relevantes, onde foi possível a obtenção da oximetria efetivamente, bem como uma análise da frequência cardíaca.

Um problema encontrado ao unificar os componentes em um único código, foi a falha ao exibir os dados da oximetria no *display* LCD. A leitura dos dados foi feita via comunicação UART, em um monitor serial no computador, que foi desenvolvido com o objetivo de depurar e otimizar o código ao longo do seu desenvolvimento.

A oximetria obtida se mostrou fiel à real quando comparada aos dados obtidos de um oxímetro comercial. Exceções a isso foram pequenos picos nas medições, o que acredita-se ser fruto de gargalos na comunicação serial utilizada para depuração do código. Futuramente, espera-se solucionar esses ruídos indesejados, com a utilização do *display* e de técnicas para reduzir a taxa de atualização dos dados exibidos, possibilitando a utilização deste.

Outro problema foi o cálculo da frequência cardíaca, devido à dificuldade de desenvolver uma função precisa de determinação do período entre dois picos na pulsação sanguínea.

De maneira geral, o desenvolvimento do protótipo foi extremamente significante quando comparado ao ponto de controle anterior, pois foi obtido com êxito a oximetria em uma plataforma nova, com implementações mais complexas que a utilizada no ponto de controle anterior, bem como foi possível verificar o funcionamento do *display* LCD separadamente, dando boas expectativas para implementações futuras.

Referências

[1] Alexander, Christian M., Lynn E. Teller, and Jeffrey B. Gross. "Principles of pulse oximetry: theoretical and practical considerations."Anesthesia & Analgesia 68.3 (1989): 368-376.

[2] “MAX30100 Pulse Oximeter and Heart-Rate Sensor IC for Wearable Health - Maxim.” Pulse Oximeter and Heart-Rate Sensor IC for Wearable Health, MAXIM Integrated, www.maximintegrated.com/en/products/sensors-and-sensor-interface/MAX30100.html.

[3] Intersecans, OXullo. Arduino-MAX30100. n.d. https://github.com/oxullo/Arduino-MAX30100

[4] Strogonovs, Raivis. Implementing Pulse Oximeter Using MAX30100. 2018, https://morf.lv/files//max30100.pdf. Accessed 1 May 2018.

[5] v, Chan and V, Underwood ,”A Single-Chip Pulsoximeter Design Using the MSP430”,Texas Instruments, Application Report SLAA274B – November 2005–Revised February 2012.

[6] Koblenski, Sam. “Lucid Mesh.” Everyday DSP for Programmers: DC and Impulsive Noise Removal, Sam Koblenski, 2015, sam-koblenski.blogspot.com/2015/11/everyday-dsp-for-programmers-dc-and.html.

[7] Schwietering, Juergen. “Filtuino - Arduino Filters.” Jayduino, Juergen Schwietering, www.schwietering.com/jayduino/filtuino.

[8] Thrasher, Philip. “C Generic Ring Buffer.” GitHub, 6 Mar. 2012, github.com/pthrasher/c-generic-ring-buffer.

[9] Eskandari, Nima. “eUSCI_B0, I2C Master multiple byte TX/RX.” TI Cloud Tools, Texas Instruments, 1 Jan. 2018, dev.ti.com/tirex/#/?link=Software/MSP430Ware/Development Tools/MSP-EXP430FR2433/Peripheral Examples/Register Level/MSP430FR2433/msp430fr243x_eusci_i2c_standard_master.c.

Apêndice

Algoritmo de Oximetria - MAX30100_PulseOximeter.h e MAX30100_PulseOximeter.c e MAX30100_SpO2Calculator.h e MAX30100_SpO2Calculator.c

```
#define SAMPLING_FREQUENCY                      100
#define CURRENT_ADJUSTMENT_PERIOD_MS            500
#define DEFAULT_IR_LED_CURRENT
    MAX30100_LED_CURR_50MA
#define RED_LED_CURRENT_START
    MAX30100_LED_CURR_27_1MA
#define DC_REMOVER_ALPHA                        0.95

#include <stdint.h>

#include "MAX30100.h"
#include "MAX30100_BeatDetector.h"
#include "MAX30100_Filters.h"
#include "MAX30100_SpO2Calculator.h"

typedef enum PulseOximeterState {
    PULSEOXIMETER_STATE_INIT,
    PULSEOXIMETER_STATE_IDLE,
    PULSEOXIMETER_STATE_DETECTING
} PulseOximeterState;

typedef enum PulseOximeterDebuggingMode {
    PULSEOXIMETER_DEBUGGINGMODE_NONE,
    PULSEOXIMETER_DEBUGGINGMODE_RAW_VALUES,
    PULSEOXIMETER_DEBUGGINGMODE_AC_VALUES,
    PULSEOXIMETER_DEBUGGINGMODE_PULSEDETECT,
    PULSEOXIMETER_DEBUGGINGMODE_PULSEPLOTTER
} PulseOximeterDebuggingMode;

PulseOximeterDebuggingMode debuggingMode;
PulseOximeterState state = PULSEOXIMETER_STATE_INIT;
LEDCurrent irLedCurrent = DEFAULT_IR_LED_CURRENT;
uint8_t redLedCurrentIndex = (uint8_t)
    RED_LED_CURRENT_START;
```

```
uint32_t tsFirstBeatDetected = 0;
uint32_t tsLastBeatDetected = 0;
uint32_t tsLastBiasCheck = 0;
uint32_t tsLastCurrentAdjustment = 0;

void (*onBeatDetected)();
bool pulseOxBegin(PulseOximeterDebuggingMode
    debuggingMode_);
void pulseOxCheckSample();
void pulseOxCheckCurrentBias();
void pulseOxUpdate();
float pulseOxGetHeartRate();
uint8_t pulseOxGetSpO2();
uint8_t pulseOxGetRedLedCurrentBias();
void pulseOxSetOnBeatDetectedCallback(void (*cb)());
void pulseOxSetIRLedCurrent(LEDCurrent
    irLedNewCurrent);
void pulseOxiShutdown();
void pulseOxResume();
```

```
#include "MAX30100_PulseOximeter.h"
#include "MAX30100.h"
#include "MAX30100_BeatDetector.h"
#include "MAX30100_BeatDetector.c"

#include "MAX30100_Filters.h"

#include "MAX30100_SpO2Calculator.h"
#include "MAX30100_SpO2Calculator.c"

bool pulseOxBegin(PulseOximeterDebuggingMode
    debuggingMode_)
{
  debuggingMode = debuggingMode_;

  bool ready = begin();

  if(!ready)
  {
    if(debuggingMode !=
    PULSEOXIMETER_DEBUGGINGMODE_NONE)
      sendString("Failed to initialize the HRM
    sensor");
    return false;
  }

  setMode(MAX30100_MODE_SPO2_HR);
    setLedsCurrent(irLedCurrent, (LEDCurrent)
    redLedCurrentIndex);

    setDCAlpha(DC_REMOVER_ALPHA, 'R');
    setDCAlpha(DC_REMOVER_ALPHA, 'I');

    state = PULSEOXIMETER_STATE_IDLE;

    return true;
}

void pulseOxCheckSample()
{
  uint16_t rawIRValue, rawRedValue;
  float irACValue, redACValue, filteredPulseValue;
  bool beatDetected;

  // Dequeue all available samples
  while(getRawValues(&rawIRValue, &rawRedValue))
  {
        irACValue = dcStepIr(rawIRValue);
        redACValue = dcStepRed(rawRedValue);

        filteredPulseValue = butterworthStep(-
    irACValue);
    beatDetected = beatDetectorAddSample(
    filteredPulseValue);

    if (beatDetectorGetRate() > 0) {
            state = PULSEOXIMETER_STATE_DETECTING;
        spO2CalcUpdate(irACValue, redACValue,
    beatDetected);
    } else if (state ==
    PULSEOXIMETER_STATE_DETECTING) {
        state = PULSEOXIMETER_STATE_IDLE;
        spO2CalcReset();
    }

    switch (debuggingMode) {
        case
    PULSEOXIMETER_DEBUGGINGMODE_RAW_VALUES:
            sendString("I-");
            sendInt((unsigned int) rawIRValue);
            sendData('\t');
            sendString("R-");
            sendInt((unsigned int) rawRedValue);
            sendData('\n');
            break;

        case
    PULSEOXIMETER_DEBUGGINGMODE_AC_VALUES:
            sendString("IRac: ");
            sendFloat(irACValue);
            sendData('\t');
            sendString("Rac: ");
            sendFloat(redACValue);
            sendData('\n');
            break;

        case
    PULSEOXIMETER_DEBUGGINGMODE_PULSEDETECT:
            sendString("R: ");
            sendFloat(filteredPulseValue);
            sendData('\t');
            sendString("TH: ");
            sendFloat(
    beatDetectorGetCurrentThreshold());
            sendData('\t');
            switch (stateBeat)
            {
                case BEATDETECTOR_STATE_INIT:
                    sendString("INIT");
                    break;
                case BEATDETECTOR_STATE_WAITING:
                    sendString("WAIT");
                    break;
                case
    BEATDETECTOR_STATE_FOLLOWING_SLOPE:
                    sendString("FSLOPE");
                    break;
                case
    BEATDETECTOR_STATE_MAYBE_DETECTED:
                    sendString("MDET");
                    break;
                case BEATDETECTOR_STATE_MASKING:
                    sendString("MASK");
                    break;
                default:
                    sendString("ERR");
                    break;
            }
            sendData('\n');
            break;

        case
    PULSEOXIMETER_DEBUGGINGMODE_PULSEPLOTTER:
            sendFloat(filteredPulseValue);
            sendData('\n');
            break;

        default:
            break;
    }

    if (beatDetected && onBeatDetected) {
        onBeatDetected();
    }
```

```
  }
}

void pulseOxCheckCurrentBias()
{
    // Follower that adjusts the red led current in order to have comparable DC baselines between
    // red and IR leds. The numbers are really magic: the less possible to avoid oscillations

    if (millis() - tsLastBiasCheck > CURRENT_ADJUSTMENT_PERIOD_MS) {
        bool changed = false;
        if (getDCW('I') - getDCW('R') > 70000 && redLedCurrentIndex < MAX30100_LED_CURR_50MA) {
            ++redLedCurrentIndex;
            changed = true;
        } else if (getDCW('R') - getDCW('I') > 70000 && redLedCurrentIndex > 0) {
            --redLedCurrentIndex;
            changed = true;
        }

        if (changed) {
            setLedsCurrent(irLedCurrent, (LEDCurrent)redLedCurrentIndex);
            tsLastCurrentAdjustment = millis();

            /*
            if (debuggingMode != PULSEOXIMETER_DEBUGGINGMODE_NONE) {
                Serial.print("I:");
                Serial.println(redLedCurrentIndex);
            }
            */
        }

        tsLastBiasCheck = millis();
    }

}

void pulseOxUpdate()
{
    update();

    pulseOxCheckSample();
    //pulseOxCheckCurrentBias();
}

float pulseOxGetHeartRate()
{
    return beatDetectorGetRate();
}

uint8_t pulseOxGetSpO2()
{
    return spO2CalcGetSpO2();
}

uint8_t pulseOxGetRedLedCurrentBias()
{
    return redLedCurrentIndex;
}

void pulseOxSetOnBeatDetectedCallback(void (*cb)())
{
    onBeatDetected = cb;
}

void pulseOxSetIRLedCurrent(LEDCurrent irLedNewCurrent)
{
    irLedCurrent = irLedNewCurrent;
    setLedsCurrent(irLedCurrent, (LEDCurrent)redLedCurrentIndex);
}

void pulseOxiShutdown()
{
    shutdown();
}

void pulseOxResume()
{
    resume();
}
```

```
#define CALCULATE_EVERY_N_BEATS 3

void spO2CalcUpdate(float irACValue, float redACValue, bool beatDetected);
void spO2CalcReset();
uint8_t spO2CalcGetSpO2();

const uint8_t spO2LUT[43] = {100,100,100,100,99,99,99,99,99,99,98,98,98,98,98,97,97,97,97,97,97,96,96,96,96,96,96,95,95,95,95,95,95,94,94,94,94,94,93,93,93,93,93};

float irACValueSqSum = 0;
float redACValueSqSum = 0;
uint8_t beatsDetectedNum = 0;
uint32_t samplesRecorded = 0;
uint8_t spO2 = 0;
```

```
#include <math.h>

#include "MAX30100_SpO2Calculator.h"

// SaO2 Look-up Table
// http://www.ti.com/lit/an/slaa274b/slaa274b.pdf

uint8_t spO2CalcGetSpO2()
{
    return spO2;
}

void spO2CalcReset()
{
    samplesRecorded = 0;
    redACValueSqSum = 0;
    irACValueSqSum = 0;
    beatsDetectedNum = 0;
    spO2 = 0;
}

void spO2CalcUpdate(float irACValue, float redACValue, bool beatDetected)
{
    irACValueSqSum += irACValue * irACValue;
    redACValueSqSum += redACValue * redACValue;
    ++samplesRecorded;

    if (beatDetected) {
        ++beatsDetectedNum;
        if (beatsDetectedNum == CALCULATE_EVERY_N_BEATS) {
            float acSqRatio = 100.0 * log(redACValueSqSum/samplesRecorded) / log(irACValueSqSum/samplesRecorded);
            uint8_t index = 0;

            if (acSqRatio > 66) {
                index = (uint8_t)acSqRatio - 66;
            } else if (acSqRatio > 50) {
                index = (uint8_t)acSqRatio - 50;
            }
            spO2CalcReset();

            spO2 = spO2LUT[index];
        }
    }
}
```

```
#ifndef MAX30100_BEATDETECTOR_H_
#define MAX30100_BEATDETECTOR_H_

#include <stdint.h>

#define BEATDETECTOR_INIT_HOLDOFF 2000     // in ms, how long to wait before counting
#define BEATDETECTOR_MASKING_HOLDOFF 200          // in ms, non-retriggerable window after beat detection
#define BEATDETECTOR_BPFILTER_ALPHA 0.6           // EMA factor for the beat period value
#define BEATDETECTOR_MIN_THRESHOLD 20             // minimum threshold (filtered) value
#define BEATDETECTOR_MAX_THRESHOLD 300            // maximum threshold (filtered) value
#define BEATDETECTOR_STEP_RESILIENCY 30           // maximum negative jump that triggers the beat edge
#define BEATDETECTOR_THRESHOLD_FALLOFF_TARGET 0.3 // thr chasing factor of the max value when beat
#define BEATDETECTOR_THRESHOLD_DECAY_FACTOR 0.99     // thr chasing factor when no beat
#define BEATDETECTOR_INVALID_READOUT_DELAY 2000     // in ms, no-beat time to cause a reset
#define BEATDETECTOR_SAMPLES_PERIOD 10            // in ms, 1/Fs


typedef enum BeatDetectorState {
    BEATDETECTOR_STATE_INIT,
    BEATDETECTOR_STATE_WAITING,
    BEATDETECTOR_STATE_FOLLOWING_SLOPE,
    BEATDETECTOR_STATE_MAYBE_DETECTED,
    BEATDETECTOR_STATE_MASKING
} BeatDetectorState;

bool beatDetectorAddSample(float sample);
float beatDetectorGetRate();
float beatDetectorGetCurrentThreshold();
bool beatDetectorCheckForBeat(float value);
void beatDetectorDecreaseThreshold();

BeatDetectorState stateBeat = BEATDETECTOR_STATE_INIT;
float threshold = BEATDETECTOR_MIN_THRESHOLD;
float beatPeriod = 0;
float lastMaxValue = 0;
uint32_t tsLastBeat = 0;
```

```
#include "MAX30100_BeatDetector.h"
#include "TimerWDT.c"
#include <stdbool.h>

#ifndef min
#define min(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })
#endif

bool beatDetectorAddSample(float sample)
{
  return beatDetectorCheckForBeat(sample);
}

float beatDetectorGetRate()
{
    if (beatPeriod != 0) {
        return 1.0 / beatPeriod * 1000.0 * 60.0;
    } else {
        return 0;
    }
}

float beatDetectorGetCurrentThreshold()
{
    return threshold;
}

void beatDetectorDecreaseThreshold()
{
    // When a valid beat rate readout is present, target the
    if (lastMaxValue > 0 && beatPeriod > 0) {
        threshold -= lastMaxValue * (1 - BEATDETECTOR_THRESHOLD_FALLOFF_TARGET) / (beatPeriod / BEATDETECTOR_SAMPLES_PERIOD);
    } else {
        // Asymptotic decay
        threshold *= BEATDETECTOR_THRESHOLD_DECAY_FACTOR;
    }

    if (threshold < BEATDETECTOR_MIN_THRESHOLD) {
        threshold = BEATDETECTOR_MIN_THRESHOLD;
    }
}


bool beatDetectorCheckForBeat(float sample)
{
    bool beatDetected = false;
    switch (stateBeat) {
        case BEATDETECTOR_STATE_INIT:
            if (millis() > BEATDETECTOR_INIT_HOLDOFF) {
                stateBeat = BEATDETECTOR_STATE_WAITING;
            }
            break;

        case BEATDETECTOR_STATE_WAITING:
            if (sample > threshold) {
                threshold = min(sample, BEATDETECTOR_MAX_THRESHOLD);
                stateBeat = BEATDETECTOR_STATE_FOLLOWING_SLOPE;
            }

            // Tracking lost, resetting
            if (millis() - tsLastBeat > BEATDETECTOR_INVALID_READOUT_DELAY) {
                beatPeriod = 0;
                lastMaxValue = 0;
            }

            beatDetectorDecreaseThreshold();
            break;

        case BEATDETECTOR_STATE_FOLLOWING_SLOPE:
            if (sample < threshold) {
                state = BEATDETECTOR_STATE_MAYBE_DETECTED;
            } else {
                threshold = min(sample, BEATDETECTOR_MAX_THRESHOLD);
            }
            break;

        case BEATDETECTOR_STATE_MAYBE_DETECTED:
            if (sample + BEATDETECTOR_STEP_RESILIENCY < threshold) {
                // Found a beat
                beatDetected = true;
                lastMaxValue = sample;
                stateBeat = BEATDETECTOR_STATE_MASKING;
                uint32_t delta = millis() - tsLastBeat;
                if (delta) {
```

```
                    beatPeriod =
    BEATDETECTOR_BPFILTER_ALPHA * delta +
                        (1 -
    BEATDETECTOR_BPFILTER_ALPHA) * beatPeriod;
                }

                tsLastBeat = millis();
            } else {
                stateBeat =
    BEATDETECTOR_STATE_FOLLOWING_SLOPE;
            }
            break;

        case BEATDETECTOR_STATE_MASKING:
            if (millis() - tsLastBeat >
    BEATDETECTOR_MASKING_HOLDOFF) {
                stateBeat =
    BEATDETECTOR_STATE_WAITING;
            }
            beatDetectorDecreaseThreshold();
            break;
    }

    return beatDetected;
}
```

Biblioteca do Sensor MAX30100 - MAX30100.h e MAX30100.c

```
#define DEFAULT_MODE
    MAX30100_MODE_HRONLY
#define DEFAULT_SAMPLING_RATE
    MAX30100_SAMPRATE_100HZ
#define DEFAULT_PULSE_WIDTH
    MAX30100_SPC_PW_1600US_16BITS
#define DEFAULT_RED_LED_CURRENT
    MAX30100_LED_CURR_50MA
#define DEFAULT_IR_LED_CURRENT
    MAX30100_LED_CURR_50MA
#define EXPECTED_PART_ID                0x11

ringBuffer_typedef(uint16_t, fifoBuffer);
fifoBuffer redBuffer, irBuffer;
fifoBuffer* redBuffer_ptr;
fifoBuffer* irBuffer_ptr;

bool begin();
bool isTemperatureReady();

uint8_t getPartId();
uint8_t readRegister(uint8_t address);
uint8_t retrieveTemperatureInteger();

float retrieveTemperature();

void writeRegister(uint8_t address, uint8_t data);
void setMode(Mode mode);
void setLedsPulseWidth(LEDPulseWidth ledPulseWidth);
void setSamplingRate(SamplingRate samplingRate);
void setLedsCurrent(LEDCurrent irLedCurrent,
    LEDCurrent redLedCurrent);
void setHighresModeEnabled(bool enabled);
void resetFifo();
void resume();
void shutdown();
void startTemperatureSampling();
void burstRead(uint8_t baseAddress, uint8_t *buffer,
     uint8_t length);
void readFifoData();
void update();
bool getRawValues(uint16_t *ir, uint16_t *red);
```

```
#include "MAX30100_Registers.h"
#include "ringbuffer.h"
#include "I2C.c"
#include "MAX30100.h"

bool begin()
{
    if(getPartId() != EXPECTED_PART_ID)
      return false;

    setMode(DEFAULT_MODE);
    setLedsPulseWidth(DEFAULT_PULSE_WIDTH);
    setSamplingRate(DEFAULT_SAMPLING_RATE);
    setLedsCurrent(DEFAULT_IR_LED_CURRENT,
    DEFAULT_RED_LED_CURRENT);
    setHighresModeEnabled(true);

    bufferInit(redBuffer,16, uint16_t);
    bufferInit(irBuffer,16, uint16_t);
    redBuffer_ptr = &redBuffer;
    irBuffer_ptr = &irBuffer;

    return true;
}


uint8_t getPartId()
{
  return readRegister(MAX30100_REG_PART_ID);
}

uint8_t readRegister(uint8_t address)
{
  uint8_t partId [1] = {0};
  I2C_Master_ReadReg(SLAVE_ADDR, address, 1);
  CopyArray(ReceiveBuffer, partId, 1);
  return partId[0];
}

void writeRegister(uint8_t address, uint8_t data)
{
  uint8_t tBuffer [1] = {0};
  tBuffer[0] = data;
  I2C_Master_WriteReg(SLAVE_ADDR, address, tBuffer,
    1);
}

void setMode(Mode mode)
{
    writeRegister(MAX30100_REG_MODE_CONFIGURATION,
    mode);
}

void setLedsPulseWidth(LEDPulseWidth ledPulseWidth)
{
    uint8_t previous = readRegister(
    MAX30100_REG_SPO2_CONFIGURATION);
    writeRegister(MAX30100_REG_SPO2_CONFIGURATION, (
    previous & 0xfc) | ledPulseWidth);
}

void setSamplingRate(SamplingRate samplingRate)
{
    uint8_t previous = readRegister(
    MAX30100_REG_SPO2_CONFIGURATION);
    writeRegister(MAX30100_REG_SPO2_CONFIGURATION, (
    previous & 0xe3) | (samplingRate << 2));
}

void setLedsCurrent(LEDCurrent irLedCurrent,
    LEDCurrent redLedCurrent)
{
    writeRegister(MAX30100_REG_LED_CONFIGURATION,
    redLedCurrent << 4 | irLedCurrent);
}

void setHighresModeEnabled(bool enabled)
{
    uint8_t previous = readRegister(
    MAX30100_REG_SPO2_CONFIGURATION);
```

```
    if (enabled) {
        writeRegister(MAX30100_REG_SPO2_CONFIGURATION, previous | MAX30100_SPC_SPO2_HI_RES_EN);
    } else {
        writeRegister(MAX30100_REG_SPO2_CONFIGURATION, previous & ~MAX30100_SPC_SPO2_HI_RES_EN);
    }
}

void resetFifo()
{
    writeRegister(MAX30100_REG_FIFO_WRITE_POINTER, 0);
    writeRegister(MAX30100_REG_FIFO_READ_POINTER, 0);
    writeRegister(MAX30100_REG_FIFO_OVERFLOW_COUNTER, 0);
}

void resume()
{
    uint8_t modeConfig = readRegister(MAX30100_REG_MODE_CONFIGURATION);
    modeConfig &= ~MAX30100_MC_SHDN;

    writeRegister(MAX30100_REG_MODE_CONFIGURATION, modeConfig);
}

void shutdown()
{
    uint8_t modeConfig = readRegister(MAX30100_REG_MODE_CONFIGURATION);
    modeConfig |= MAX30100_MC_SHDN;

    writeRegister(MAX30100_REG_MODE_CONFIGURATION, modeConfig);
}

float retrieveTemperature()
{
    int8_t tempInteger = readRegister(MAX30100_REG_TEMPERATURE_DATA_INT);
    float tempFrac = readRegister(MAX30100_REG_TEMPERATURE_DATA_FRAC);

    return tempFrac * 0.0625 + tempInteger;
}

uint8_t retrieveTemperatureInteger()
{
    return readRegister(MAX30100_REG_TEMPERATURE_DATA_INT);
}

bool isTemperatureReady()
{
    return !(readRegister(MAX30100_REG_MODE_CONFIGURATION) & MAX30100_MC_TEMP_EN);
}

void startTemperatureSampling()
{
    uint8_t modeConfig = readRegister(MAX30100_REG_MODE_CONFIGURATION);
    modeConfig |= MAX30100_MC_TEMP_EN;

    writeRegister(MAX30100_REG_MODE_CONFIGURATION, modeConfig);
}

void burstRead(uint8_t baseAddress, uint8_t *buffer, uint8_t length)
{
  I2C_Master_ReadReg(SLAVE_ADDR, baseAddress, length);
  CopyArray(ReceiveBuffer, buffer, length);
}

void readFifoData()
{
  uint8_t buffer[MAX30100_FIFO_DEPTH*4];
    uint8_t toRead, i;
    uint16_t redWrite, irWrite;

    toRead = (readRegister(MAX30100_REG_FIFO_WRITE_POINTER) - readRegister(MAX30100_REG_FIFO_READ_POINTER)) & (MAX30100_FIFO_DEPTH-1);

    if(toRead)
    {
      burstRead(MAX30100_REG_FIFO_DATA, buffer, 4 * toRead);

      for (i=0 ; i < toRead ; ++i)
      {
        irWrite = (uint16_t)((buffer[i*4] << 8) | buffer[i*4 + 1]);
        redWrite = ((buffer[i*4 + 2] << 8) | buffer[i*4 + 3]);
        bufferWrite(irBuffer_ptr,irWrite);
        bufferWrite(redBuffer_ptr,redWrite);
      }
    }

}

void update()
{
  readFifoData();
}

bool getRawValues(uint16_t *ir, uint16_t *red)
{
  if(!isBufferEmpty(irBuffer_ptr) && !isBufferEmpty(redBuffer_ptr))
  {
    bufferRead(irBuffer_ptr, *ir);
    bufferRead(redBuffer_ptr, *red);
    return true;
  }
  else
    return false;
}
```

Filtros para o MAX30100 - Removedor DC e Butterworth - MAX30100_Filters.h

```
// http://www.schwietering.com/jayduino/filtuino/
// Low pass butterworth filter order=1 alpha1=0.1
// Fs=100Hz, Fc=6Hz
float v[2];

float butterworthStep(float x) //class II
{
  v[0] = v[1];
  v[1] = (2.452372752527856026e-1 * x)
     + (0.50952544949442879485 * v[0]);
  return
     (v[0] + v[1]);
}

// http://sam-koblenski.blogspot.de/2015/11/everyday-dsp-for-programmers-dc-and.html
float alphaRed, alphaIr;
float dcwRed = 0, dcwIr = 0;

void setDCAlpha(float alpha_, char c)
{
  switch(c)
```

```
  {
    case 'R':
      alphaRed = alpha_;
      break;
    case 'I':
      alphaIr = alpha_;
      break;
    default:
      break;
  }
}

float getDCW(char c)
{
  switch(c)
  {
    case 'R':
      return dcwRed;
    case 'I':
      return dcwIr;
    default:
      return 0;
  }
}

float dcStepRed(float xRed)
{
  float olddcwRed = dcwRed;
  dcwRed = (float)xRed + alphaRed * dcwRed;

  return dcwRed - olddcwRed;
}

float dcStepIr(float xIr)
{
  float olddcwIr = dcwIr;
  dcwIr = (float)xIr + alphaIr * dcwIr;

  return dcwIr - olddcwIr;
}
```

Algoritmo para display LCD - LCD.c

```
#include <msp430fr2433.h>

#define BTN BIT3
#define LCD_OUT P2OUT
#define LCD_DIR P2DIR
#define D4 BIT4
#define D5 BIT5
#define D6 BIT6
#define D7 BIT7
#define RS BIT0
#define E  BIT1
#define DADOS 1
#define COMANDO 0
#define CMND_DLY 1000
#define DATA_DLY 1000
#define BIG_DLY  3000
#define CLR_DISPLAY  Send_Byte(1, COMANDO, BIG_DLY)
#define POS0_DISPLAY Send_Byte(2, COMANDO, BIG_DLY)
#define POS1_DISPLAY Send_Byte(0xC0, COMANDO,
    BIG_DLY)

void Atraso_us(volatile unsigned int us);
void Send_Nibble(volatile unsigned char nibble,
    volatile unsigned char dados, volatile unsigned
    int microsegs);
void Send_Byte(volatile unsigned char byte, volatile
     unsigned char dados, volatile unsigned int
    microsegs);
void Send_Data(volatile unsigned char byte);
void Send_String(char str[]);
void Send_Int(int n);
void Send_Float(volatile float var_float);
void InitLCD(void);

void InitLCD(void)
{
  unsigned char CMNDS[] = {0x02, 0x01, 0x28, 0x0E};
  unsigned int i;
  // Atraso de 10ms para o LCD fazer o boot
  Atraso_us(10000);
  LCD_DIR |= D4+D5+D6+D7+RS+E;
  Send_Nibble(0x03, COMANDO, CMND_DLY);
  for(i=0; i<4; i++)
    Send_Byte(CMNDS[i], COMANDO, CMND_DLY);
  CLR_DISPLAY;
  POS0_DISPLAY;
}

void Atraso_us(volatile unsigned int us)
{
    volatile unsigned int k = 0;
    while (k < 16)
    {
        TA1CCR0 = us-1;
        TA1CTL = TASSEL_2 + ID_3 + MC_3 + TAIE;
        while((TA1CTL & TAIFG)==0);
        TA1CTL = TACLR;
        TA1CTL = 0;
        k++;
    }
}

void Send_Nibble(volatile unsigned char nibble,
    volatile unsigned char dados, volatile unsigned
    int microsegs)
{
  LCD_OUT |= E;
  LCD_OUT &= ~(RS + D4 + D5 + D6 + D7);
  LCD_OUT |= RS*(dados==DADOS) +
    D4*((nibble & BIT0)>0) +
    D5*((nibble & BIT1)>0) +
    D6*((nibble & BIT2)>0) +
    D7*((nibble & BIT3)>0);
  LCD_OUT &= ~E;
  Atraso_us(microsegs);
}

void Send_Byte(volatile unsigned char byte, volatile
     unsigned char dados, volatile unsigned int
    microsegs)
{
  Send_Nibble(byte >> 4, dados, microsegs/2);
  Send_Nibble(byte & 0xF, dados, microsegs/2);
}

void Send_Data(volatile unsigned char byte)
{
  Send_Byte(byte, DADOS, DATA_DLY);
}

void Send_String(char str[])
{
  while((*str)!='\0')
  {
    Send_Data(*(str++));
  }
}

void Send_Int(int n)
{
  int casa, dig;
  if(n==0)
  {
    Send_Data('0');
    return;
  }
  if(n<0)
  {
    Send_Data('-');
    n = -n;
  }
```

```
103   for(casa = 10000; casa>n; casa /= 10);
104   while(casa>0)
105   {
106     dig = (n/casa);
107     Send_Data(dig+'0');
108     n -= dig*casa;
109     casa /= 10;
110   }
111 }
112
113 void Send_Float(volatile float var_float)
114 {
115     volatile int var_int;
116
117     if (var_float < 0) // se for negativo
118     {
119         var_float = var_float * (-1); // multiplica
      por -1
120         Send_Data('-'); // imprime sinal negativo
121     }
122
123   var_int = (int) var_float; // converte para
      inteiro
124   Send_Int(var_int); // envia parte inteira
125
126   Atraso_us(2);
127   Send_Data('.'); // envia o "."
128
129   var_float = (var_float - var_int)*100; //
      multiplica a parte residual nao inteira
130   var_int = (int) var_float; // converte as duas
      primeiras casas decimais em inteiro
131   Send_Int(var_int); // envia as duas primeiras
      casas decimais
132 }
```