UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Pedro Pappis Bandeira e Vitória Son Kantorski

SISTEMAS LÓGICOS PROGRAMÁVEIS:
DESENVOLVIMENTO DE UM OSCILOSCÓPIO UTILIZANDO O ARDUÍNO MEGA 2560

Santa Maria, RS
2024

SUMÁRIO

1 INTRODUÇÃO

O *software* IDE do Arduino fornece um *plotter* dedicado, porém com funcionalidades limitadas à representação gráfica dos valores convertidos pelo ADC. Este projeto tem como o objetivo o desenvolvimento de um *plotter* com comportamento similar a um osciloscópio utilizando o microcontrolador ATMega2560 em conjunto com a placa de desenvolvimento Arduino MEGA, juntamente a uma interface para visualização dos dados obtidos.

A ferramenta desenvolvida expande as capacidades apresentadas pelo IDE, simulando um osciloscópio que pode ser replicado por qualquer pessoa que tenha acesso a uma placa Arduino e a um computador. Suas funcionalidades incluem acoplamento DC e AC, análise de frequência (FFT) e medição de valores médios e máximos. Além disso, os dados obtidos podem ser exportados em formato .csv para análises adicionais em outros softwares.

2 OBJETIVOS

2.0.1 Obejtivo Geral

Desenvolver um software capaz de fazer a leitura de níveis de tensão utilizando o microcontrolador ATMega2560 e uma interface para a visualização dos dados obtidos.

2.0.2 Objetivos específicos

- Estudar lógica de programação utilizando o Arduino.
- Desenvolver conhecimentos da plataforma Arduino, especialmente a interface USART.
- Desenvolver conhecimentos acerca de elementos básicos de conversores A/D.
- Produzir um dispositivo capaz de mensurar a tensão de um circuito e fornecer uma visualização precisa de sua forma de onda.

3 METODOLOGIA

A elaboração do projeto foi feita priorizando pesquisas acerca da capacidade dos dispositivos utilizados e o desenvolvimento de funcionalidades similares a osciloscópios utilizados em laboratório a partir destas. A metodologia de desenvolvimento do projeto segue o fluxograma da Figura **??** abaixo.

	Descrição	**Período**
1	Elaboração da proposta de projeto utilizando o ATmega 2560	09 a 10/04
2	Analisar as capacidades do Arduino Mega, teorica e experimentalmente	10 a 16/04
2	Início do desenvolvimento do *software*	16 a 26/04
3	Teste em simulação	26/04 a 01/05
4	Desenvolvimento do software de interface	02 a 10/05
5	Teste em simulação do projeto final	10 a 13/05
6	Implementação, correções ou melhorias	13 a 17/05
7	Testes físicos em laboratório	13 a 17/05
8	Elaboração do relatório para entrega	10/04 a 18/05
9	Entrega do relatório final e apresentação	21/05/2024

Tabela 1 – Cronograma de atividades

Inicialmente, foram realizadas simulações através das atividades disponibilizadas em aula de Sistemas Lógicos e Programáveis dos comandos e análises possíveis com o ATMega2560 em plataforma Wokwi. A partir disto, foram analisadas as funcionalidades principais dos osciloscópios comerciais e definidas as necessárias ao projeto (Tabela 2).

Função	**Prioridade**
Comunicação *Arduino* e computador	Obrigatória
Interface de visualização dos dados adquiridos	Obrigatória
Trigger e cursores	Obrigatória
Ajuste de *offset*	Obrigatória
Aquisição de métricas (PP, Amplitude, Frequência)	Obrigatória
Controles Físicos	Alta
FFT (Fast Fourier Transform)	Baixa

Tabela 2 – Prioridade de funcionalidades

A partir das informações teóricas e técnicas, iniciou-se o desenvolvimento do código em plataforma Wokwi, onde foi definido as portas de entrada e saída, a configuração do ADC e USART, os controles de número de samples, tempo de aquisição de dados e *step* (determinada pelo delay entre a aquisição de dados). Concomitante ao desenvolvimento do código, foram realizadas simulações em laboratório NUPEDEE para verificação do seu funcionamento. De acordo com os resultados da simulação, eram realizadas modificações no código.

A implementação se deu quando as funcionalidades obrigatórias citadas anteriormente foram plenamente desenvolvidas, mesmo com divergências não planejadas. Utilizando do conversor AD integrado do chip ATMega2560, a leitura de valores de tensão e transmissão destes via UART segue o fluxograma abaixo.

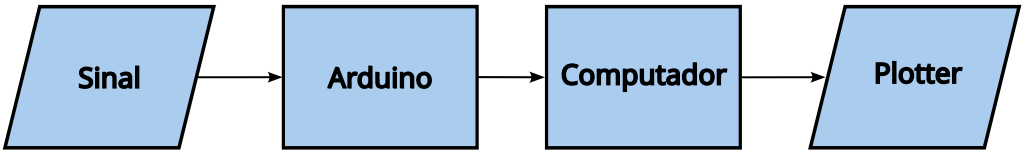

Figura 1 – Fluxograma do Projeto

3.1 CONSIDERAÇÕES DE PROJETO

A conversão correta de um sinal analógico para uma representação discreta, neste caso, digital, recai sobre o Teorema de Nyquist.

$$f_s > 2 \times f_{max} \tag{3.1}$$

Como descrito na equação 3.1, a frequência de aquisição (*Sample Rate*) deve ser o dobro da frequência do sinal amostrado. Isto impõem uma limitação à quantidade de sinais que podem ser amostrados pelo ADC.

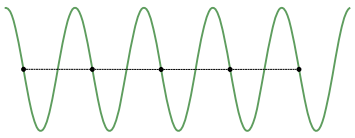

Figura 2 – Sample Rate = 1

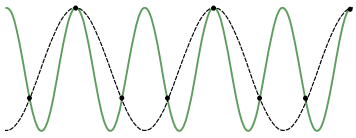

Figura 3 – Sample Rate = 1.5

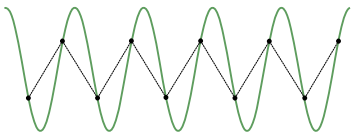

Figura 4 – Sample Rate = 2

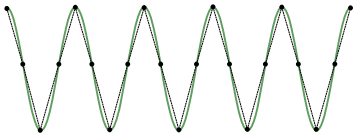

Figura 5 – Sample Rate = 4

Figura 6 – Sample Rate = 15

Por exemplo, caso a frequência de amostragem seja igual a frequência do sinal, teremos o caso da Figura 2, uma linha reta, pois, se o sinal for periódico, todas as amostras retornarão o mesmo valor.

Já na Figura 3, com 1.5X a frequência do sinal, é possível fazer a interpolação dos pontos para reconstrução do sinal, mas perdemos a frequência original.

Ou seja, quanto mais amostras, mais próximo do sinal real a representação será, mas, para a realidade e para o projeto, buscamos o menor número de samples que nos permita reconstruir o sinal da forma mais fiel possível.

3.1.1 Projeto

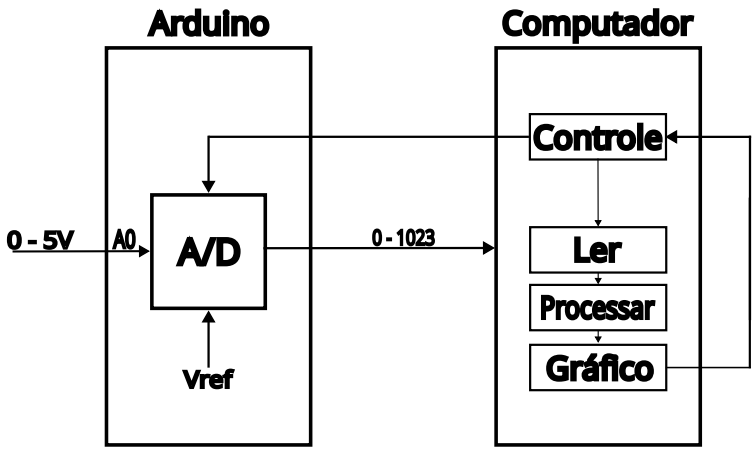

Figura 7 – Modelo do projeto

3.2 ADC

Do *data sheet*, temos que a frequência do *clock* do ADC deve estar entre 50 KHz e 200 KHz e, se a resolução for menor que 10 *bits* a frequência pode ser maior que 200 KHz para um maior *sample rate*.

Aqui está o primeiro *Trade-off*, em troca de uma resolução maior, limitamos a frequência dos sinais que podem ser adquiridos pelo ADC, visto que:

$$10 \text{ bits de precisão} = 2^{10} = 1024 \text{ Níveis de tensão}$$

Sabendo que o escopo de operação é de 0 a 5 V:

$$\text{Precisão} = \frac{5\ V}{1024} = 0.00488V = 4.88mV$$

Desta forma, usando um *prescaler* que divide a frequência do *clock* do processador por 128:

$$CLK_{ADC} = \frac{CLK_{SYS}}{N} = \frac{16MHz}{128} = 125KHz$$

Ainda no *data sheet*, é especificado que cada conversão toma 13 ciclos de *clock* do ADC para ser concluída, desta forma, o tempo necessário para cada conversão é

$$13 \times \frac{1}{125KH} \approx 0.1ms$$

O prescaler é configurado pelos bits ***ADPS[0:2]*** do registrador ***ADCRSA*** exposto

na Figura 8.

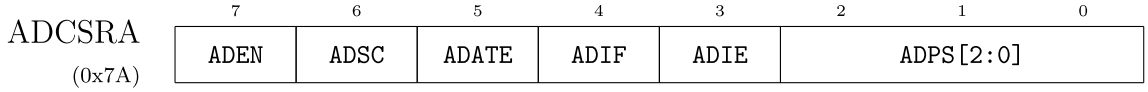

Figura 8 – Registrador ADCRSA

Os valores dos bits ***ADPS[0:2]*** para diferentes valores do *prescaler* estão expostos na Tabela *23-5 ADC Prescaler Selections*. Desta forma, foi escolhido ***ADPS[111]*** ou N = 128 pois possibilitava a maior frequência de *clock*, como discutido anteriormente.

Abaixo está o código de inicialização do conversor

```
void ADC_init() {
   /* Configura o prescaler do ADC para 128 */
   ADCSRA |= (1 << ADPS2);
   ADCSRA |= (1 << ADPS1);
   ADCSRA |= (1 << ADPS0);
   /* AVcc como tensao de referencia */
   ADMUX = (1 << REFS0);
   /* Inicia o conversor */
   ADCSRA |= (1 << ADEN);
}
```

O registrador ***ADMUX*** (Figura 9) também foi utilizado

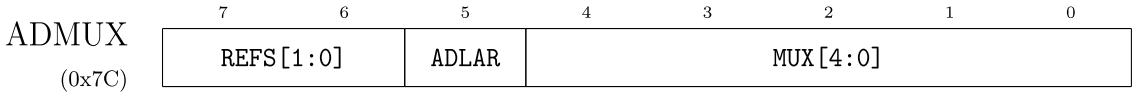

Figura 9 – Registrador ADMUX

Segundo a Tabela *23-3. Voltage Reference Selections for ADC*, ***REFS1*** = 0 e ***REFS0*** = 1 resultam na tensão de referência AV_{CC} ou $V_{REF} = 5V$.

Desta forma, na Figura 10 está descrito uma representação gráfica do processo de aquisição de amostras do ADC.

O número de samples é definido pelo usuário e limitado pela memória RAM do dispositivo usado, neste caso, um Arduino UNO, com 2 kBytes de memória. Como será discutido posteriormente.

Após definir o intervalo de amostragem, calcula-se o tempo entre cada amostra.

Em outras palavras, determina-se o *delay* necessário para que todas as amostras sejam coletadas dentro do período estabelecido. O calcule segue a equação 3.2

$$\Delta T_{\text{amostras}} = \frac{\text{Tempo de Amostragem}}{\text{Número de Amostras}} - \text{Tempo de aquisição} \tag{3.2}$$

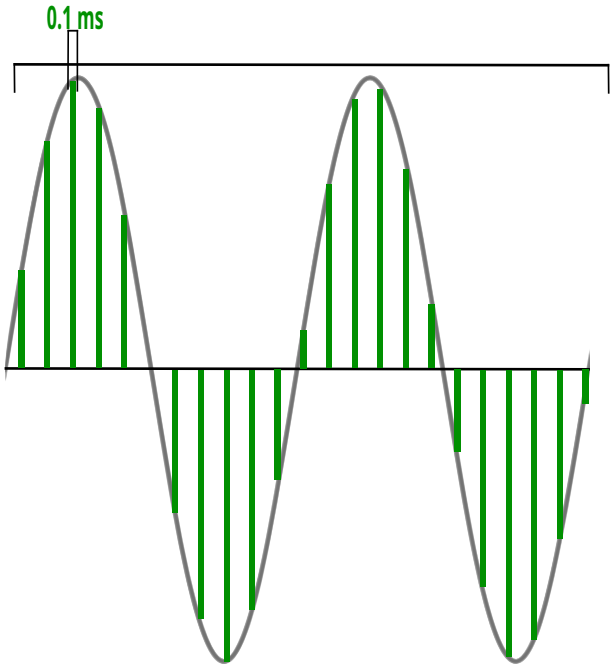

Figura 10 – Enter Caption

Por exemplo, para um tempo de aquisição de *200ms* e 500 amostras:

$$\Delta T_{\text{amostras}} = \frac{200}{500} - 0.1 = 0.3\ ms$$

No código em C e com os valores de tempo sendo trabalhado em μS:

```
delayAmostraBurstuSec = ((duracaoBurstmSec * 1000) /
                        numAmostrasBurst) - 100;
```

Com estas informações é possível calcular a banda do osciloscópio, ou seja, qual é a máxima frequência de amostragem:

$$B = \frac{1}{0.1\ ms} = 10\ KHz$$

Um valor significativamente inferior se comparado aos osciloscópios comerciais, que alcançam frequências de amostragem na ordem dos giga-hertz. De qualquer forma, de acordo com o teorema de nyquist, já discutido nas considerações do projeto, a maior frequência de sinal que pode ser amostrada pelo osciloscópio é:

$$f_{\text{MAX}} = \frac{10\ KHz}{2} = 5\ KHz$$

Ainda, o Arduino MEGA possui 8 kBytes de memória RAM, o Arduino UNO, possui 2. Sabendo que o tamanho médio de cada sample adquirida é de 2 bytes (*unsigned int*), considerando um espaço livre de 500 bytes.

$$\text{MEGA} = \frac{8192 - 500}{2} = 3846 \text{ samples}$$
$$\text{UNO} = \frac{2048 - 500}{2} = 774 \text{ samples}$$

3.3 USART

Como mencionado anteriormente, cada conversão do ADC leva, em média, 0,1 ms. Portanto, se forem realizadas 100 aquisições, o tempo total gasto será de 10 ms. A questão é: como transmitir 100 valores, ou N valores, da maneira mais rápida possível?

A *USART*, ou, neste caso, a *UART*, possui sua taxa de transmissão medida em *BAUD*, que equivale a bps (bits por segundo). Segundo a Table 19-12. *Examples of UBRRn Settings for Commonly Used Oscillator Frequencies*, o valor máximo para uma frequência de 16 MHz do CLK_{SYS} é 2 Mbps, 2 milhões de bits por segundo, taxa obtida usando o modo *Double Speed Operation*.

As samples são armazenadas em uma variável do tipo *unsigned int* de tamanho 2 bytes, ou 16 bits, anteriormente foi visto que as samples podem variar de 0 a 1023, desta forma, a *string* que será transmitida pela USART tem o formato [0-1023]+$\backslash n$, ou seja, um total de, no máximo, 5 caracteres *ASCII UTF-8*.

- De 0 - 9 $\rightarrow$ 1$\times$ 8+8 = 16 bits
- De 10 - 99 $\rightarrow$ 2$\times$ 8+8 = 24 bits
- De 100 - 999 $\rightarrow$ 3$\times$ 8+8 = 32 bits
- De 1000 - 1023 $\rightarrow$ 4$\times$ 8+8 = 40 bits

$$\text{Tamanho Médio} = \frac{10 \times 16 + 90 \times 24 + 900 \times 32 + 24 \times 40}{1024} \approx 32 \text{ bits} \tag{3.3}$$

Ou seja, com um tamanho médio de 32 bits e uma taxa de transmissão de 2Mbps o tempo necessário para a transmissão de 500 amostras é dado pela equação 3.4

$$T_{\text{Trans.}} = \frac{500 \times 32}{2000000} = 0.008\ s = 8\ ms \tag{3.4}$$

Em comparação, com uma taxa de 115200 bps, na equação 3.5 é notável a importância de operar na frequência mais alta possível, visto que, no exemplo de aquisição de

500 samples em 100 ms de duração, com a transmissão mais baixa, passaríamos 139% do tempo na transmissão e 50% na aquisição do ADC, sem contar o *delay*.

$$T_{\text{Trans.}} = \frac{500 \times 32}{1115200} = 0.139\ s = 139\ ms \tag{3.5}$$

Os registradores responsáveis pela configuração da USART0 são ***UBRR0***, ***UCSR0A***, ***UCSR0B*** e ***UCSR0C***. Estes são apresentados na Figura 11.

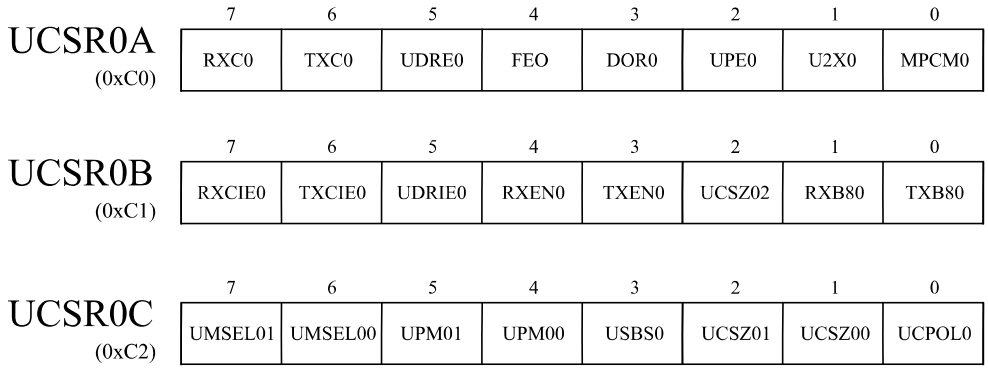

Figura 11 – Registrador USART0

```
#define MYUBRR F_CPU/8/BAUD-1
   void uart_init(unsigned int ubrr) {
   /* Configura a taxa BAUD */
   UBRRH = (unsigned char)(ubrr>>8);
   UBRRL = (unsigned char)ubrr;

   UCSR0A = (1 << U2X0);

   /* Ativa Transmissor Receptor */
   UCSRB = (1<<RXEN)|(1<<TXEN);
   /* Formato de dado : 8data, 1stop bit */
   UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
   }
```

C code example page 206 [ATmega640/V-1280/V-1281/V-2560/V-2561/V [DATASHEET]]

Para calcular **UBRR0** usamos a equação 3.6, levando em conta **U2X0 = 1**

$$UBRRn = \frac{f_{\text{SYS}}}{8\text{BAUD}} - 1 \tag{3.6}$$

Para um BAUD de 2 Mbps:

$$UBBR0 = \frac{16000000}{8 \times 2000000} - 1 = 0 \tag{3.7}$$

Utilizamos a *USART* no modo assíncrono, **UMSEL0[1:0]=00** (conforme a *Table 22-*

4 UMSELn Bits Settings). Quadros no formato 8-N-1 **UCSZ0[2:0]=011** (de acordo com a *Table 22-7 UCSZn Bits Settings*) para representar 8 bits de dados. Além disso, configuramos os bits **UPM0[1:0]=00** (conforme a *Table 22-5 UPMn Bits Settings*) para desativar a paridade e o bit **USBS0=0** (conforme a *Table 22-6 USBS Bit Settings*) para um único bit de parada. Configurações feitas de forma implícita já que são *default* no Arduino.

Ainda dos *C code examples*, foram retiradas as seguintes funções:

```
void uart_transmit(unsigned char data) {

   while (!(UCSR0A & (1 << UDRE0)));
   UDR0 = data;
}

unsigned char uart_receive(void) {
   while (!(UCSR0A & (1 << RXC0)));
   return UDR0;
}

void uart_print(const char *str) {
   while (*str) {
      uart_transmit(*str++);
   }
}
```

3.4 TIMER

O projeto é altamente dependente na exatidão dos *delays*, logo, foi usado o *Timer1* para gerar as interrupções sem depender de bibliotecas como *delay.h*.

Escolhemos o *Timer1* de 16 bits pois estamos trabalhando com valores em microssegundos, ou seja, permite contar até 65,536 *ms*, tempo mais que suficiente para a faixa temporal do nosso osciloscópio.

```
   void timer1_init() {
   TCCR1A = 0;
   TCCR1B = (1 << CS11);   // prescaler para 8
   TIMSK1 = (1 << OCIE1A); // Ativa a comparacao
   TCNT1 = 0;                // Reseta o contador
   sei();                     // Ativa as interrupcoes
}
```

Na Figura 12 estão os principais registradores de controle de Timer1

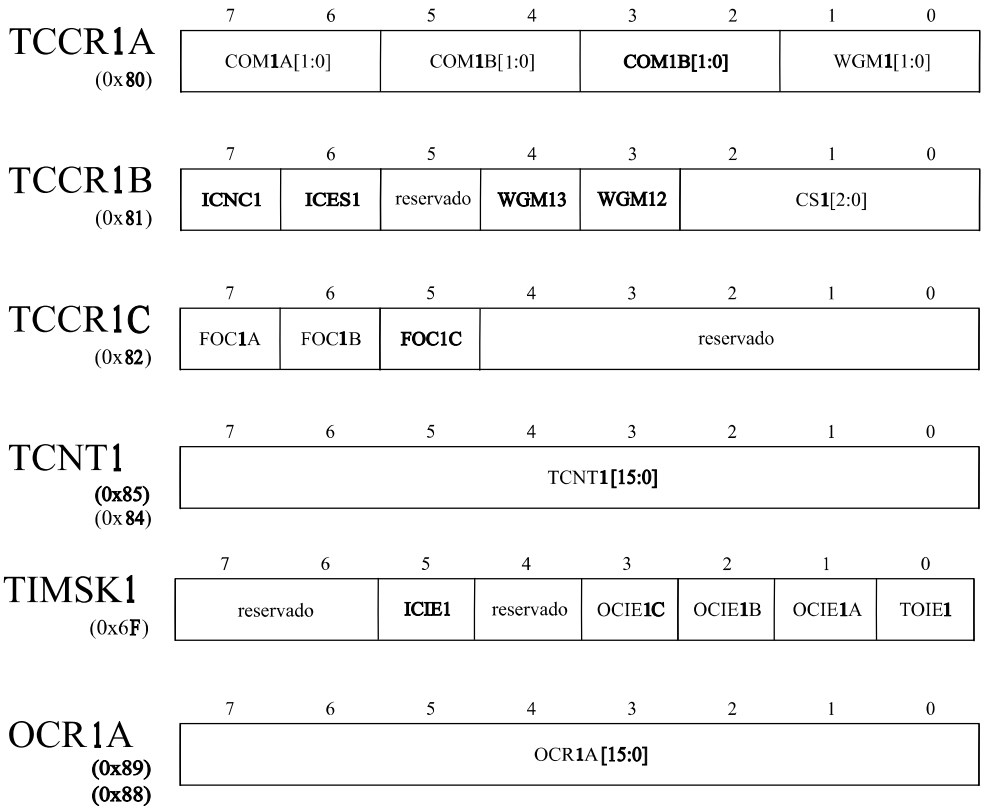

Figura 12 – Registradores Timer 1

O valor de comparação do registrador **OCR1A** são passados de forma interativa pelo código. com o *prescaler* = 8, o temporizador é incrementado a cada 0.5 μS, desta forma, como nosso objetivo é 1μS, basta multiplicar o valor do *delay* por 2, assim, em código:

```
   ISR(TIMER1_COMPA_vect) {
   delay_flag = 1;
}

void delayuseconds(uint16_t us) {
   cli();
   delay_flag = 0;
   TCNT1 = 0;
   OCR1A = us * 2;
   sei();
   while (!delay_flag);
}
```

Na função *delayuseconds()* passamos o *delay* calculado com a equação 3.2 como uma *uint16_t* pois **OCR1A** tem 16 bits. Este valor é, então, multiplicado por dois, como já discutido. Na Figura 13 está apresentado o fluxograma desta função.

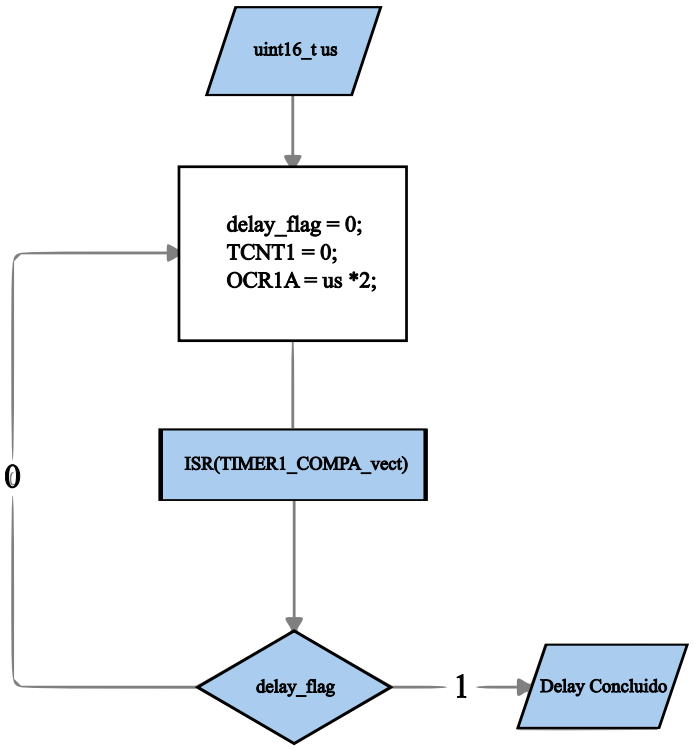

Figura 13 – Registradores Timer 1

Com isto, garantimos o funcionamento de todas as partes do osciloscópio, garantimos *delays* precisos, uma alta taxa de transmissão pela *USART* e utilizamos toda a capacidade do ADC para nossas aquisições, agora nos resta a interface com o *PC*.

4 INTERFACE PC

A interface foi feita utilizando o *framework windows forms (WinForms)* e, portanto, escrita em C# , não cabe a este trabalho demonstrar todas suas funcionalidades, logo, apenas as partes mais importantes serão abordadas.

4.0.1 Comunicação com Arduino

A comunicação com o Arduino é realizada via comandos com prefixos, mais especificamente, o prefixo ***B*** é acompanhado por um número que especifica a duração do período de aquisição em milissegundos. Ex: **B100** = 100 *ms*

O segundo prefixo é o ***S***, que determina o número de samples a serem adquiridas. Ex: ***S500*** = 500 samples.

Estes comandos são interpretados pelo Arduino da seguinte forma:

```
while (1) {
    if (UCSR0A & (1 << RXC0)) {
        char c = uart_receive();
        if (c == '\n') {
            receiveString[receiveIndex] = '\0';
            if (receiveString[0] == 'S') {
                numBurstSamples = atoi(&receiveString[1]);
            } else if (receiveString[0] == 'B') {
                burstDurationmSec = atol(&receiveString[1]);
                GrabBurstandSend();
            }
            receiveIndex = 0;
        } else {
            receiveString[receiveIndex++] = c;
            if (receiveIndex >= sizeof(receiveString) - 1) {
                receiveIndex = sizeof(receiveString) - 1;
            }
        }
    }
}
```

4.0.2 Processamento dos Dados

Após o recebimento das aquisições denotadas pela linha "FIM", os dados são processados pelo computador.

As principais funções de processamento são:

```
   public void ParsarCSVBurst()
{
   int i = 0;

   foreach (string s in this.listaStringEntrada)
   {
      string valString = s.TrimEnd('\r', '\n');

      if (int.TryParse(valString, out this.arrayIntEntrada[i])) { }
      else
      {
         this.arrayIntEntrada[i] = 0;
      }

      i++;
   }
}
public void EscalarValoresBurst(int intervaloEscala, double Vref)
{
   double intervaloDouble = Convert.ToDouble(intervaloEscala);

   for (int i = 0; i < this.arrayIntEntrada.Length; i++)
   {
      double amostraDouble =
         Convert.ToDouble(this.arrayIntEntrada[i]);
      this.arrayDoubleProcessada[i, 1] = Vref * amostraDouble /
         intervaloDouble;
   }
}
public void ZerarTemposBurst(double intervaloAmostraMSec)
{
   double tempoAmostra = 0.0;

   for (int i = 0; i < this.arrayIntEntrada.Length; i++)
   {
      this.arrayDoubleProcessada[i, 0] = tempoAmostra;
```

```
        tempoAmostra += intervaloAmostraMSec;
    }
}
```

A função *ParsarCSVBurst()* transforma a *string* enviada pelo arduino em uma *array* de inteiros.

EscalarValoresBurst() transforma a *array* de inteiros em uma *array* de *Doubles* para transformar os valores convertidos pelo ADC em tensões de acordo com a tensão de referência.

ZerarTemposBurst() é responsável pelo cálculo de tempo entre cada amostra, como discutido anteriormente, a fim de uma transmissão mais rápida e menor uso de RAM, a escolha foi abandonar a precisão de enviar o tempo e optar por uma aproximação calculada.

Também existem as funções responsáveis pelo cálculo das métricas como: Valor Máximo, Média, Frequência, FFT e etc. Mas essas serão omitidas deste relatório.

Ainda, temos toda a infraestrutura para o plot das amostras e calculo das próximas métricas de tempo de aquisição para o Arduino, estas, também, serão omitidas. Porém, é necessário ressaltar que este programa leva em consideração um *buffer* para que não haja envio de comando enquanto qualquer uma das partes está ocupada.

5 RESULTADOS

Foram obtidas as medidas de alguns sinais no laboratório do NUPEDEE utilizando um gerador de sinais. Foram medidos tanto com nosso osciloscópio e um comercial. As imagens do osciloscópio comercial serão omitidas. As imagens estão em anexo.

Os sinais gerados tinham um *offset* de 2 volts e uma amplitude de 1 V_{PP}. É interessante notar que as frequências medidas não batem com as apresentadas pelo gerador de sinais. Possíveis motivos: Ausência de um *Trigger* e Imprecisão nos calculos do tempo.

6 CONCLUSÃO

Este trabalho mostra que é possível criar um osciloscópio, ainda que falho em muitos aspectos, com uma ferramenta barata como as placas Arduino.

Ainda há muitas possibilidades para aprimoramentos, como um sistema de trigger, tanto para bordas, subida e decida, como para níveis de tensão.

Acreditamos que, dentro das limitações já apresentadas, nosso código navega de forma eficiente a fim de alcançar os melhores resultados possíveis.

Como ferramenta, o osciloscópio desenvolvido se mostra bastante poderoso e, especialmente, para pessoas no nível de *Hobbysta* ou para aplicações rápidas é um excelente custo-benefício, é possível afirmar, então, que está não é uma ferramenta de precisão.

De qualquer forma, o desenvolvimento do projeto contribui tanto para solidificar a teoria sobre microcontroladores e, além disso, muitos conceitos desenvolvidos durante o curso de engenharia elétrica.

REFERÊNCIAS

[1] Giovani Baratto. Conversor AnálogoDigital.

[2] Giovani Baratto. Configurando e Usando a USART0 no Modo Assíncrono.

[3] Giovani Baratto. Temporizador/Contador 0.

[4] Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V: Datasheet (fev. de 2014). 2549. Rev. Q. Atmel Corporation

[5] https://www.youtube.com/@EETechStuff