Universidade Federal de Juiz de Fora
Programa de Pós-Graduação em Engenharia Elétrica
Mestrado em Engenharia Elétrica

Segundo Trabalho

Aluno: Thiago Duque Saber de Lima

Juiz de Fora
13 de novembro de 2023

Sumário

1 Introdução

O segundo trabalho da disciplina tem como objetivo implementar um conversor digital analógico utilizando o modulo PWM do TIVA.

Para isso, serão considerados como pré-requisitos:

- Enviar um sinal a ser sintetizado pela porta serial via Python;
- Utilizar um filtro de reconstrução para remover harmônicas de alta ordem;
- Realizar uma discussão sobre banda, resolução e frequência do PWM, demonstrando resultados práticos;

Dessa maneira, foi estabelecido um sistema com o objetivo de receber do python um sinal a ser sintetizado. No TIVA, esse sinal foi recebido através do periférico de UART modulado através de uma onda PWM de frequência 32 kHz. Para isso, foi utilizado o periférico de PWM. Foi utilizado um periférico de timer para alterar o valor do PWM de acordo com a frequência do sinal enviado. Á saída desse sinal foi ligada a um filtro passa-baixa, sendo então conectada a um osciloscópio. Na Figura 1 está ilustrado o funcionamento do sistema.

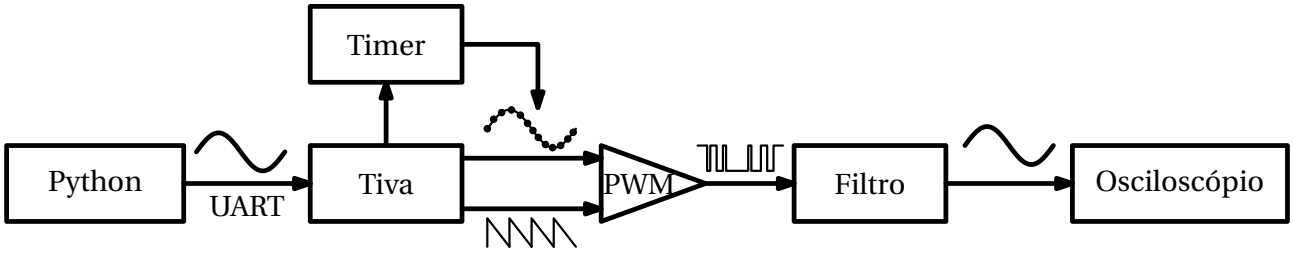

Figura 1: Configuração para funcionamento do sistema.

2 Python

Para o programa em Python, foi modificado o arquivo do primeiro trabalho para somente fazer o envio dos dados. Dessa forma, é enviado somente um vetor com porcentagens de duty cycle, conforme visto em Código 1.

Código 1: Sinal enviado pelo Python

```
my_signal = [67,  68,  70,  72,  74,  76,  78,  79,  81,  82,
   84,
        86,  87,  89,  90,  91,  93,  94,  95,  96,  96,  97,
        98,  98,  99,  99,  99,  99, 100,  99,  99,  99,  99,
        98,  98,  97,  96,  96,  95,  94,  93,  91,  90,  89,
        87,  86,  84,  82,  81,  79,  78,  76,  74,  72,  70,
        68,  67,  65,  62,  61,  59,  57,  55,  54,  52,  51,
        48,  47,  46,  44,  43,  42,  40,  39,  38,  37,  36,
        35,  35,  34,  34,  33,  33,  33,  33,  33,  33,  33,
        34,  34,  35,  35,  36,  37,  38,  39,  40,  42,  43,
        44,  46,  47,  48,  51,  52,  54,  55,  57,  59,  61,
        62,  65]
```

Tal sinal tende a recriar uma senoide, conforme ilustrado na Figura 2.

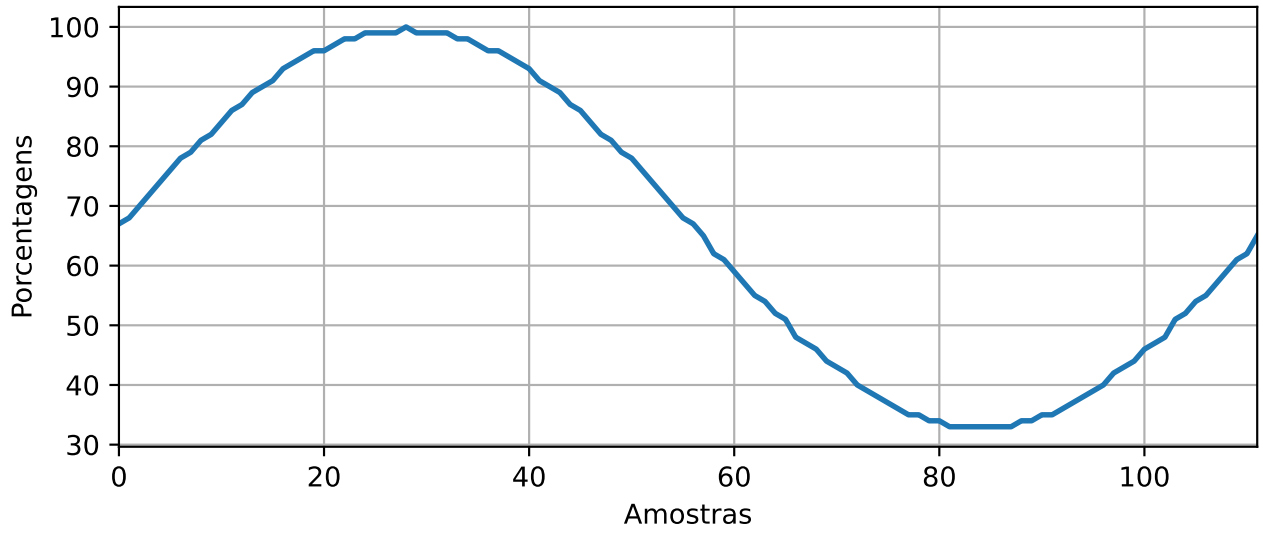

Figura 2: Sinal enviado do python para o Tiva

O código completo em python pode ser visto nos Anexos.

3 UART

Para a configuração do periférico de UART foi utilizado o código exposto em Código 2. Tal código habilita o periférico de UART e GPIO, configurando os pinos PA0 e PA1 como RX e TX, respectivamente. Além disso, esse código faz a configuração de baudrate, stop bit e bit de paridade.

Código 2: Configuração do periferico de UART

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_UART0)){ }
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOA)){ }
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1);
UARTConfigSetExpClk(
        UART0_BASE, systemClock, 115200,
        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
           UART_CONFIG_PAR_NONE));
```

Para fazer o recebimento do dados vindo do programa em python, foi utilizado o trecho de código exposto em Código 3, que tem como objetivo preencher um vetor com as porcentagens de duty cycle desejadas.

Código 3: Recebendo código do python

```
while(i < SIGNAL_LENGTH){
    if (UARTCharsAvail(UART0_BASE)){
        duty[i] = UARTCharGet(UART0_BASE);
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ((SystemClockFreq/
           freq_pwm)*25*duty[i])/100);
        i = i+1;
    }
}
```

4 PWM

Para a configuração do periférico de PWM foi utilizado o código exposto em Código 4. Tal código habilita o periférico de PWM, utilizando o gerador de PWM 0 e PWM1, saindo no pino PF1.

Código 4: Configuração do periferico de PWM

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF));
GPIOPinConfigure(GPIO_PF1_M0PWM1);
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM0));
PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN |
   PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 25*SystemClockFreq/freq_pwm
   );
uint32_t ui32MyDuty = 25*((SystemClockFreq/freq_pwm)*50.0)
   /(100.0);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ui32MyDuty);
PWMGenEnable(PWM0_BASE, PWM_GEN_0);
PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);
```

Para fins de teste, foi utilizado gerado um PWM com frequência de 32 kHz e duty cycle de 50%. É possível verificar essa onda através da Figura 3, em que é possível comprovar o funcionamento correto.

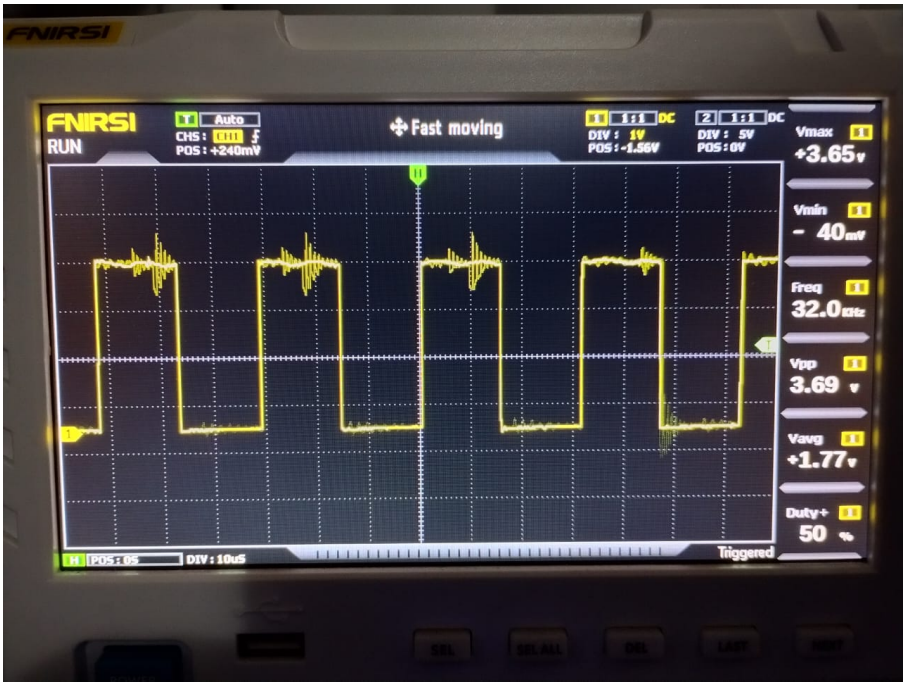

Figura 3: Geração de onda PWM com 32 kHz e 50% de duty cycle.

5 TIMER

Para fazer a mudança dos valores de duty cycle, foi configura um periferico de timer para chamar uma interrupção com um periodo de tempo pré-determinado, de acordo com a frequência do sinal enviado pelo python e número de amostras. A configuração do periferico de timer foi feita utilizando o código exposto em Código 5. Tal código configura o periférico de timer e de interrupção.

Código 5: Configuração do periferico de TIMER/Interrupção

```
uint16_t ui16SignalFreq = 500;
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
uint32_t ui32Period = SystemClockFreq/(ui16SignalFreq *
   SIGNAL_LENGTH);
TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);
IntEnable(INT_TIMER0A);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntMasterEnable();
TimerEnable(TIMER0_BASE, TIMER_A);
```

6 Filtro

Para esse trabalho, foi utilizado o mesmo filtro projetado no trabalho 1. Na Figura 4 está ilustrado o filtro passa-baixas projetado, sendo esse um filtro sallen-key de quarta ordem.

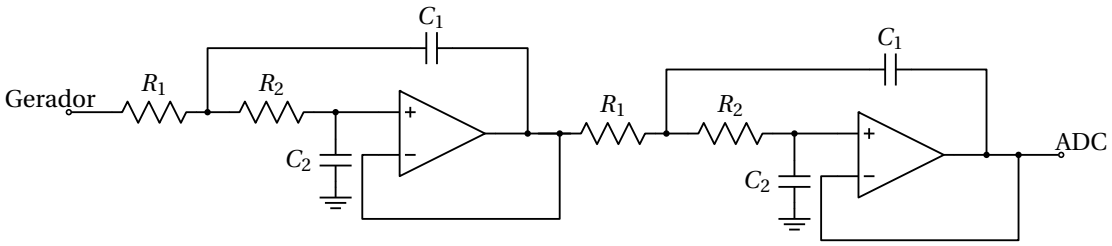

Figura 4: Filtro de quarta-ordem.

O sistema foi projetado para ter uma frequência de amostragem de 12 kHz. Sendo assim, foi projetado um filtro para ter frequência de corte em 3,3862 kHz com os componentes expostos na Tabela 1.

Componente	Valor	Unidade
R_1	$4,7$	kΩ
R_2	$4,7$	kΩ
C_1	10	nF
C_2	10	nF

Tabela 1: Valores dos componentes selecionados.

7 Resultados

Na Figura 5 está demonstrando a saída do pino PF1 com a razão cíclica variando, conforme o esperado.

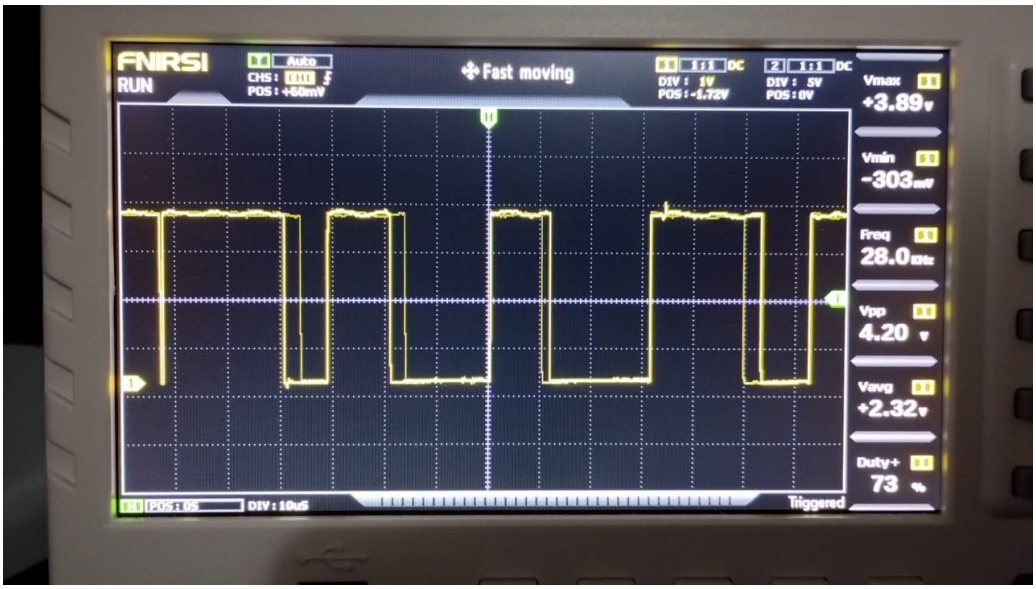

Figura 5: PWM com razão cíclica variando.

Na Figura 6 está ilustrado o sinal pós saída do filtro. É possível reparar que o sinal possuí o shape desejado, porém não tem a frequência esperada, decorrente de possível erro de configuração do timer.

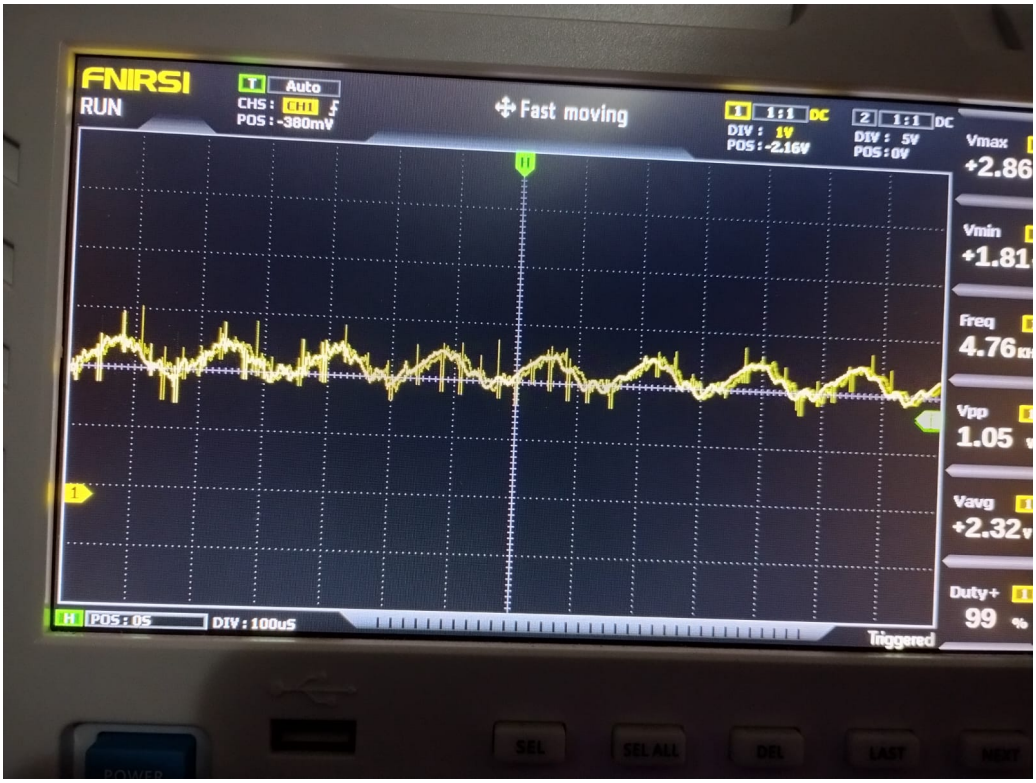

Figura 6: Saída do filtro.

8 Análise sobre banda, resolução e frequência

Para a geração do PWM, é utilizado um modulo timer de 16 bits de resolução. O PWM funciona a partir de um valor pré definido **LOAD**, que seria um valor máximo para um contador decrescente de ciclos de clock. O valor de *duty-cycle* é definido através do valor de um comparador (COMPA). Na Figura 7 está ilustrado como funciona a geração do PWM.

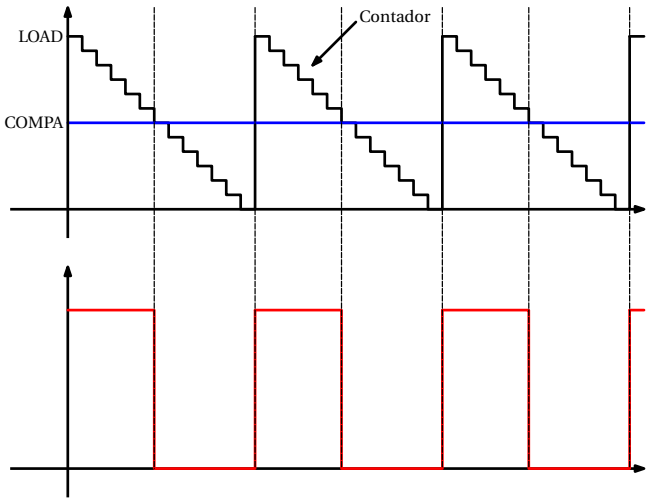

Figura 7: Geração do PWM.

Considerando que foi utilizado o clock interno de 120 MHz (ou seja, sem a utilização de

um *pre-scaler*), a frequência máxima do PWM é limitada através do valor máximo da variável LOAD. Utilizando um valor de LOAD da forma 2^n, podemos calcular a frequência do PWM através da Equação 1, em que foi utilizado o valor de $n = 16$.

$$f_{max} = \frac{f_{clock}}{2^n} = \frac{120000000}{65536} = 1831{,}05\,\text{Hz} \tag{1}$$

Dessa forma, é possível perceber que o valor máximo de frequência alcançado pelo módulo PWM é definido pelo clock do sistema, pelo valor de *pre-scaler* e pelo valor da variável LOAD. Na Tabela 2 estão expostos os valores de frequência máxima para cada valor de resolução do modulo timer, utilizando o *pre-scaler* de 1.

Clock	Resolução	Frequência máxima	Clock	Resolução	Frequência Máxima
120 MHz	16	1831,05 Hz	80 MHz	16	1220,70 Hz
120 MHz	15	3662,10 Hz	80 MHz	15	2441,40 Hz
120 MHz	14	7324,21 Hz	80 MHz	14	4882,81 Hz
120 MHz	13	14 648,43 Hz	80 MHz	13	9765,625 Hz
120 MHz	12	29 296,87 Hz	80 MHz	12	19 531,25 Hz
120 MHz	10	117 187,50 Hz	80 MHz	10	78 125,00 Hz
120 MHz	8	468 750,00 Hz	80 MHz	8	312 500,00 Hz

Tabela 2: Frequência máxima para valores de clock e resolução.

Como dito anteriormente, a variável LOAD dita a frequência do PWM, sendo quanto maior a variável de LOAD, menor o valor de frequência que se pode obter do PWM. A variável de LOAD está diretamente ligada também aos níveis discretos de resolução do duty-cycle. Utilizando uma variável de **LOAD** de 16 bits (LOAD = 65536), o comparador COMPA pode possuir 65536 níveis discretos para o duty-cycle. Considerando um duty-cycle de 0 a 1, teríamos um valor de $1.52 \cdot 10^{-5}$ por bit. Quando reduzimos o valor de LOAD, ou seja, aumentamos o valor de frequência, diminuímos os valores discretos que o duty-cycle pode assumir. Para um LOAD = 1024, o duty-cycle varia de $9,76 \cdot 10^{-4}$ por bit.

Sendo assim, é possível perceber que a banda, resolução e frequência são variáveis amarradas. Para ser possível obter um maior valor de frequência mantendo o valor da variável LOAD é necessário aumentar a frequência do clock. Para aumentar os valores discretos possiveis para o duty-dycle é necessário ou aumentar a frequência do clock ou da variavel LOAD.

Anexos

Código 6: Arquivo microcontrolador

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_adc.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"

   // Inclui a biblioteca de timer
#include "driverlib/interrupt.h"

   // Inclui a biblioteca de interrupt
#include "driverlib/adc.h"

   // Inclui a biblioteca de adc
#include "driverlib/uart.h"

   // Inclui a biblioteca de uart
#include "driverlib/udma.h"

   // Inclui a biblioteca de DMA
#include "driverlib/pwm.h"

   // Inclui a biblioteca de PWM

#define SIGNAL_LENGTH 112

uint32_t freq_pwm = 32000;
uint32_t SystemClockFreq;
float SystemClockPeriod;
uint8_t i = 0, flagChangeDuty = 0;
uint8_t duty[SIGNAL_LENGTH];

// Interrup para alterar o valor de pwm
void Timer0IntHandler(void)
{
    // Clear the timer interrupt
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ((SystemClockFreq/
       freq_pwm)*25*duty[i])/100);
```

```
    i = i + 1;

       // Incrementa i
    i = i % 112;

       // Faz o buffer circular
}

int main(void)
{
    uint32_t systemClock = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ
       | SYSCTL_OSC_MAIN
    | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);

       // Define o clock do sistema (clock interno, 120 MHz)

    SystemClockFreq = SysCtlClockGet();

       // Recebe a frequencia do clock interno
    SystemClockPeriod = 1/SystemClockFreq;

       // Calcula o periodo do clock


    // ----------------------------------------- Configura a
       UART -----------------------------------------
        SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
        while(!SysCtlPeripheralReady(SYSCTL_PERIPH_UART0)){ }
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
        while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOA)){ }
        GPIOPinConfigure(GPIO_PA0_U0RX);
        GPIOPinConfigure(GPIO_PA1_U0TX);
        GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1)
           ;
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
        GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0 |
           GPIO_PIN_1);
        UARTConfigSetExpClk(
                UART0_BASE, systemClock, 115200,
                (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                   UART_CONFIG_PAR_NONE));

    // ----------------------------------------- CONFIGURA PWM
       -----------------------------------------
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

```
    // Habilita o periferico de GPIOF
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF));
                                                                //
     Espera habilitar o GPIOF
GPIOPinConfigure(GPIO_PF1_M0PWM1);

    // Configura o pino PF1 para PWM1
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);

    // Configura o pino PF1 para PWM1
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);

    // Habilita o periferico de pwm
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_PWM0));

    // Verifica e espera ate habilitar o PWM1
PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN |
    PWM_GEN_MODE_NO_SYNC);                                  //
    Configura o modulo 0, gerador de PWM 0, para trabalhar no
    modo dowm
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 25*SystemClockFreq/
    freq_pwm);                                              //
    Configura o periodo do PWM (LOAD = SystemClockFreq/
    freq_pwm)
uint32_t ui32MyDuty = 25*((SystemClockFreq/freq_pwm)*50.0)
    /(100.0);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ui32MyDuty);
                                    // Coloca o DutyCycle de
    0.50/
PWMGenEnable(PWM0_BASE, PWM_GEN_0);

    // Habilita o Timer do gerador 0
PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);

    // Habilita a saida do pwm



// ---------------------------------------- RECEBE O SINAL
    ----------------------------------------
while(i < SIGNAL_LENGTH){
    if (UARTCharsAvail(UART0_BASE)){
        duty[i] = UARTCharGet(UART0_BASE);
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ((
            SystemClockFreq/freq_pwm)*25*duty[i])/100);
        i = i+1;
```

```
        }
    }
    i = 0;

    // ------------------------------------ CONFIGURA TIMER/
       INTERRUPT ------------------------------------
    uint16_t ui16SignalFreq = 500;

       // Frequencia do sinal recebido
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

       // Habilita o periferico de timer
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

       // Configura o timer como periodico
    uint32_t ui32Period = SystemClockFreq/(ui16SignalFreq *
       SIGNAL_LENGTH);                                          //
       Calculo do contador da interrupcao
    TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);

       // Configura o contador da interrupcao
    IntEnable(INT_TIMER0A);

       //
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

       //
    IntMasterEnable();

       //
    TimerEnable(TIMER0_BASE, TIMER_A);

       //

    while(1){

    }

}
```

Código 7: Arquivo main.py

```
from realtimeplot import RealTimePlot
import matplotlib.pyplot as plt


freq = 2000
```

```
my_signal = [67,  68,  70,  72,  74,  76,  78,  79,  81,  82,
   84,
        86,  87,  89,  90,  91,  93,  94,  95,  96,  96,  97,
        98,  98,  99,  99,  99,  99, 100,  99,  99,  99,  99,
        98,  98,  97,  96,  96,  95,  94,  93,  91,  90,  89,
        87,  86,  84,  82,  81,  79,  78,  76,  74,  72,  70,
        68,  67,  65,  62,  61,  59,  57,  55,  54,  52,  51,
        48,  47,  46,  44,  43,  42,  40,  39,  38,  37,  36,
        35,  35,  34,  34,  33,  33,  33,  33,  33,  33,  33,
        34,  34,  35,  35,  36,  37,  38,  39,  40,  42,  43,
        44,  46,  47,  48,  51,  52,  54,  55,  57,  59,  61,
        62,  65]


port = 'COM14'
baudrate = 115200
print(len(my_signal))
comm = RealTimePlot(port = port, baudrate = baudrate, pong =
   False)
comm.getSignal(signal=my_signal)
comm.sendSignal()
```

Código 8: Arquivo realtimeplot.py

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import numpy as np
from communicate import Communicate


class RealTimePlot():
    _fig_width_cm = 18/2.4
    _fig_height_cm = 7/2.4
    _SAMPLE_TIME = 8.333e-5
    _WINDOW_TIME = 10e-3
    __i = 0

    def __init__(self, port: str, baudrate: int, pong = False):
        '''
        Classe para fazer o plot em tempo real.
        :param str port: porta de comunica  o com o
           microcontrolador
        :param int baudrate: baudrate da comunica  o
        :param bool pong: configura o envio de um sinal para o
           microcontrolador
        '''
        self._pong = pong
```

```
        self._y = []
        self._t = np.arange(0, -self._WINDOW_TIME, -self.
            _SAMPLE_TIME)
        self._fig, self._axs = plt.subplots(figsize=(self.
            _fig_width_cm, self._fig_height_cm), nrows = 1, ncols
            = 1)
        self._communicate = Communicate(port=port, baudrate=
            baudrate)

    def initAnimate(self) -> None:
        '''
        M todo para iniciar o plot em tempo real.
        '''
        print("teste")
        self._ani = FuncAnimation(self._fig, self._update,
            interval = 86)
        print("teste 2")


    def getSignal(self, signal : list) -> None:
        '''
        M todo para pegar o sinal que deve ser reenviado.
        :param list signal: sinal que deve ser reenviado
        '''
        self._signal = signal

    def sendSignal(self):
        for signal in self._signal:
            self._communicate.send_int_to_uart(signal)

    def _update(self, fig) -> None:
        print("Teste 3")
        try:
            if self._pong == True:
                print(f'Enviando a amostra {self.__i}: {self.
                    _signal[self.__i]}')
                self._communicate.send_int_to_uart(self._signal[
                    self.__i])
                self.__i = (self.__i + 1) % len(self._signal)
            data = self._communicate.read_int_from_uart()
            self._updateData(data)
            self._updatePlot()
        except Exception as e:
            print(f"Erro: {e}")

    def _updateData(self, y : int) -> None:
```

```
        '''
        M todo que atualiza o vetor de dados recebidos.
        '''
        # self._y.insert(0, y)
        # print(self._y)
        self._y = np.concatenate((y, self._y))
        if len(self._y) > len(self._t):
            self._y = self._y[:len(self._t)]
        # print(f'y: {self._y}')

    def _updatePlot(self) -> None:
        '''
        M todo que atualiza o gr fico em tempo real
        '''
        try:
            self._axs.cla()
            print(self._y)
            self._axs.plot(self._t[:len(self._y)], self._y,
               linewidth = 2, color = "tab:blue")
            self._axs.set_xlim([-self._WINDOW_TIME,0])
            # self._axs.set_ylim([60, 130])
            self._axs.grid()
            print("teste")
        except:
            print("Erro ao plotar gr fico")
```

Código 9: Arquivo communicate.py

```
from realtimeplot import RealTimePlot
import matplotlib.pyplot as plt


freq = 2000
my_signal = [67,  68,  70,  72,  74,  76,  78,  79,  81,  82,
   84,
        86,  87,  89,  90,  91,  93,  94,  95,  96,  96,  97,
        98,  98,  99,  99,  99,  99, 100,  99,  99,  99,  99,
        98,  98,  97,  96,  96,  95,  94,  93,  91,  90,  89,
        87,  86,  84,  82,  81,  79,  78,  76,  74,  72,  70,
        68,  67,  65,  62,  61,  59,  57,  55,  54,  52,  51,
        48,  47,  46,  44,  43,  42,  40,  39,  38,  37,  36,
        35,  35,  34,  34,  33,  33,  33,  33,  33,  33,  33,
        34,  34,  35,  35,  36,  37,  38,  39,  40,  42,  43,
        44,  46,  47,  48,  51,  52,  54,  55,  57,  59,  61,
        62,  65]
```

```
port = 'COM14'
baudrate = 115200
print(len(my_signal))
comm = RealTimePlot(port = port, baudrate = baudrate, pong =
   False)
comm.getSignal(signal=my_signal)
comm.sendSignal()
```