SEL0630 - PROJETOS EM SISTEMAS EMBARCADOS

Prática 2: Introdução a programação em sistemas embarcados com GPIO, Sensores, Periféricos e Computação Paralela

Resumo

Introdução a programação em Python, com uso de condicionais, laços de repetição, funções, conversão de tipo, importação de bibliotecas, manipulação de erros, e uso de gpio no Python da Raspberry, sensores, gerência de processos, threads, e computação paralela..

Conceitos importantes:
Type casting, Timer, time, Import, Try Except, if, for, while, GPIO, Virtual environment, sensors, PWM, threads, mutex, semaphores, deadlock, round-robin, parallel computing, multicore CPU..

Parte 1

Objetivo

O objetivo desta prática é a familiarização dos conceitos básicos que tange a linguagem de programação Python, abrangendo os conceitos de importação de bibliotecas, uso de estruturas condicionais e laços de repetição, funções definidas pelo usuário, métodos de conversão de tipo e métodos de manipulação de erros, além disso, o uso de GPIO da rasp a fim de demonstrar uma simples aplicação de acesso aos componentes de hardware da placa a partir do Python.

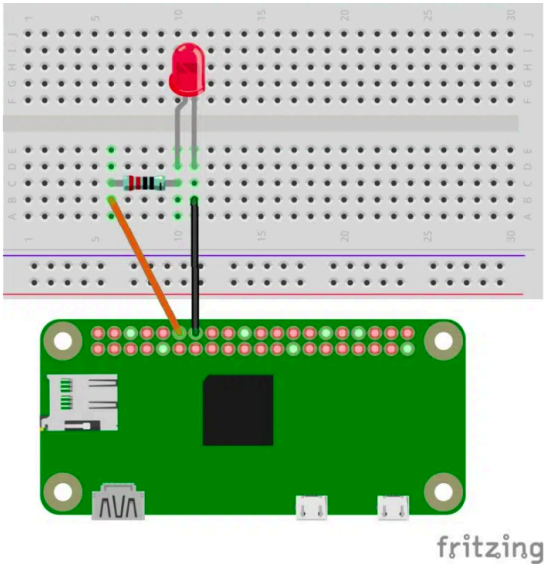
Aplicação a ser desenvolvida

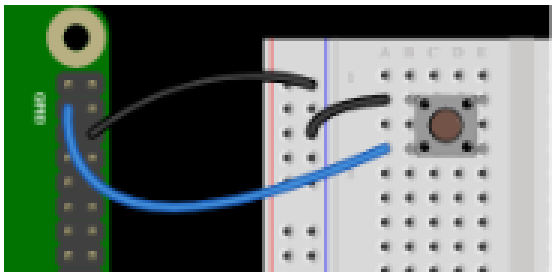
Portanto, a partir dos conceitos apresentados acima, a prática consiste em realizar um script em Python, responsável por receber um dado de entrada via terminal, e partir dele, realizar uma contagem regressiva, o convertendo para o formato de minutos e segundos, e ao final da contagem, acender um LED na SBC e escrever uma mensagem no terminal indicando o fim da contagem.

Antes de executar qualquer script e realizar a instalação de bibliotecas python, é importante que se crie um ambiente virtual. Desta forma, as bibliotecas estarão instaladas somente dentro deste ambiente e não em toda a máquina.

Um ambiente virtual python é responsável por isolar um diretório do computador do restante da máquina, desta forma, todos os comandos de instalação python executados dentro dele não afetarão o restante dos projetos, evitando conflitos em projetos que usem bibliotecas diferentes. Portanto, como primeira atividade a ser executada, deve-se criar um ambiente virtual em um diretório para o projeto, e acessá-lo, a partir disso pode-se iniciar os scripts.

Para a conexão do LED a placa, utilize a conexão demonstrada abaixo:





O pinout com descrição e numeração de cada um dos pinos pode ser consultado a partir da internet a partir do site de pinout da rasp (ou também por meio do comando “pinout” no terminal).

Motivação

A manipulação de scripts em Python é de extrema importância para sistemas operacionais embarcados, principalmente os baseados em sistemas Linux, uma vez que, na maioria das distribuições linux, o Python já vem pré-instalado, como é o caso do Raspberry Pi OS, que pôde ser consultado na prática anterior a sua versão instalada.

Além disso, em sistemas operacionais embarcados, devido a ampla utilização do Python em diversas áreas, criou-se uma comunidade grande em torno do desenvolvimento e otimização de bibliotecas e pacotes em Python, a fim de garantir que os projetos executados em Python possam ser competitivos com outras linguagens de programação, mantendo ainda a sua facilidade de aprendizado. Com isto, a aplicação deste conhecimento em sistemas embarcados garante que o processo de aprendizado seja facilitado, e possua otimizações feitas pela comunidade para que possa ser executado tão eficientemente quanto outras linguagens de programação compiladas.

Outra importância da programação em Python está no uso de servidores, onde o Python pode gerenciar várias etapas do processo de uso e criação de servidores WEB. Além da área de processamento de imagens, onde muitos dos algoritmos modernos são feitos pensados em Python a fim de garantir que o acesso a estes algoritmos seja facilitado, dada a grande base de usuários. Em

ambas as aplicações podem ser aproveitadas em sistemas embarcados, principalmente os baseados em sistemas operacionais e SoC, onde geralmente se utiliza dos conceitos de redes e servidores, além de visão computacional.

Roteiro

- Por padrão do Raspberry Pi OS, o Python normalmente já se encontra instalado. Para confirmar verifique com: python - -version (caso precisar instalar: sudo apt-get install python)
- O comando python abre um terminal desta linguagem com a versão instalada. Para sair do terminal Python digitar quit()
- Para criar e editar um arquivo no terminal usar nano nome_arquivo.py. Para executar: python nome_arquivo.py (é possível usar uma IDE para otimizar a programação. Na Raspberry Pi, dois editores estão disponíveis, digitando no terminal os nomes: thonny e geany;
- Para instalação de pacotes específicos do Python, usa-se o comando pip (gerenciador de pacotes), que também por padrão já se encontra instalado: pip - -version (para instalar: sudo apt-get install python3-pip)
- Antes de se utilizar o Python na Raspberry Pi, deve-se criar um ambiente virtual.
- Instale o venv a partir do terminal via "python3 -m pip install - -user virtualenv" ou "sudo apt install python3-venv -y" (a diferença entre usar *sudo apt instal*l é que esta busca pacotes oficiais para o sistema operacional, ao passo que *pip install* instala pacotes específicos para Python em nível de projeto - caso uma opção não funcione, testar a outra).
- A partir disto, pode-se iniciar a sua criação e ativação do ambiente virtual (venv):
    - No diretório home (/hom/sel ou ~) crie um venv com o nome sendo os dois últimos dígitos do seu NUSP (em caso de dupla, utilize os dois últimos dígitos de cada integrante: python3 -m venv XXYY)
    - Ative o venv no diretório por meio do comando source XXYY/bin/activate, lembre-se de alterar o valor XXYY pelo nome criado para o venv.
    - Para confirmar a ativação, verifique se o nome do ambiente virtual XXYY aparece antes da identificação: *'(XXYY)sel@raspberrypi:~$*' - isto indica que você acessou o ambiente virtual - para sair digitar o comando deactivate (e sempre que for acessar o ambiente, usar: source XXYY/bin/activate)
    - Uma vez ativado, pode-se testar se o Python está funcional com o comando python e quit() para sair.
    - Sendo assim, será possível realizar a instalação de pacotes restritos ao ambiente virtual.
    - Primeiramente, acesse a pasta: cd XXYY e verifique os pacotes por meio do comando pip freeze.
    - Instale as bibliotecas **gpiozero** e **RPi.GPIO** por meio do comando pip3 install <>; verifique novamente os pacotes instalados por meio do comando pip freeze.
- Faça a montagem do circuito para o blink LED, conforme imagem acima (o circuito foi desenvolvido no software **Fritzing** - é possível instalar na Raspberry Pi ou nos PCs do laboratório para montagem do circuito - foi disponibilizado o instalador no e-Disciplinas - no entanto, não será necessário para esta atividade) utilizando os componentes, jumpers e a

Protoboard. Posteriormente, realizar a montagem do circuito para leitura de um botão, também conforme imagem acima.

- Escreva programas em Python, conforme exemplos a serem demonstrados na aula, importando bibliotecas Python gpiozero e RPi.GPIO, configurando pinos GPIO, e:
    - Programando o LED para piscar com intervalo inicial de 0,5 segundos. Note que o programa de exemplo utiliza "laço condicional infinito". O primeiro exemplo usa a biblioteca gpiozero e o segundo, por outro lado, usa a RPi.GPIO;
    - Programando a leitura de um botão configurando a detecção de eventos por meio da biblioteca RPi.GPIO.

Entregas:

- Usando a biblioteca RPi.GPIO, escreva um programa (ou faça a adaptação dos programas de exemplo) para acender um LED ao pressionar o botão e apagar quando soltar o botão, tendo como base os programas de exemplos de blink LED e detecção de borda. Teste o programa sem alterar a montagem na Protoboard, mantendo os mesmos pinos GPIO para LED e botão.
- Por fim, escreva um programa em Python de forma que um LED acenda somente após a contagem regressiva de um valor de tempo de entrada, em segundos, digitado no terminal. Implementando as funcionalidades descritas anteriormente (vide trecho destacado em amarelo acima: "**type casting**"; uso do comando "**print**" para imprimir mensagem de erro quando de um valor de entrada errado; comando "**input**" para receber o valor de entrada que deve ser inteiro (**int**) digitado no terminal, por meio do qual será feita a contagem regressiva para acender o LED). OBS. Ao final da contagem, o LED deverá somente acender e permanecer nesta condição (não é necessário apagá-lo), assim como uma mensagem deverá ser impressa indicando o final da contagem. Consulte a documentaçao das bibliotecas Python e exemplos na internet.
    - Como está sendo utilizado a entrada de dados, no caso a variável de tempo a ser contado, é importante que se utilize métodos de manipulação de erros fornecidos pela linguagem, e métodos de estruturas condicionais, para verificar se os valores inseridos são realmente números e se são valores de tempo possíveis.
    - Em caso de inserção de um valor inválido, o script não pode retornar um erro que gere uma saída forçada, e sim deve somente apresentar na tela qual tipo de erro e pedir novamente ao usuário uma entrada adequada (por exemplo: "o valor digitado deve ser um número" ou "o número deve ser positivo").Dica: usar Try - Except - ValueError. A execução da contagem regressiva, a fim de manter a organização e modularização do código, deve ser feita em uma função, que recebe o parâmetro de tempo já testado do usuário.
    - Atente-se a identação do código, uma vez que em Python, a indentação não somente serve para auxiliar na legibilidade do texto, mas também para definir se uma linha está no escopo de uma função, parâmetro ou laço. Utilize a biblioteca time para gerar a base de tempo da contagem regressiva, de forma a garantir que o intervalo de tempo de contagem seja preciso.
    - Utilize Type casting a fim de garantir que o valor a ser contado esteja no formato desejado, e teste sobre ele as condições de erro. Para impressão em terminal da contagem, utilize formatação de strings, de forma a demonstrar o valor de minutos

e segundos no padrão de tempo MM : SS (por exemplo: “00:30” se valor digitado no terminal for 30; ou “01:10” se o valor foi 70 etc.). Dica: usar a função “divmod” para separação; estrutura {:02d}:{:02d}'.format(minutos,segundos) para formatação; e a estrutura end='\r' no print para atualização dos valores da contagem na mesma linha.

Critérios de avaliação da Parte 1

1 - Uso do ambiente virtual, instalação das bibliotecas Python e elaboração dos programas de exemplo demonstrados em aula a ser verificado por meio do histórico de comandos do terminal.
2 - Montagem dos circuitos e implementação dos programas de exemplo na Raspberry Pi
3 - Elaboração dos programas em Python solicitados que realiza o acionamento de uma saída (LED) a partir da mudança de estado/borda/evento de uma entrada (botão); bem como o programa que realiza a contagem regressiva para acender um LED (uso de typecasting, manipulação de erros, input, print, biblioteca RPi.GPIO etc.) e sua implementação prática na Raspberry Pi.

Formato de entrega

Enviar na tarefa somente 3 arquivos: o arquivo “.txt” contendo salvo o histórico de comandos usados no terminal Linux para execução da atividade, e os scripts em Python “.py” dos programas solicitados (acionamento do LED com botão ; e contagem regressiva) com as linhas de código comentadas.
Devido ao número limitado de kits, realizar as atividades práticas em duplas . Para entrega, basta que apenas um(a) dos/as integrantes realize o upload no e-Disciplinas. Não esquecer, no entanto, de identificar devidamente no documento de entrega o nome e NUSP de todos(as) envolvidos(as).

Bibliografia

- GPIO Zero
- LOM3260 — Curso de Computação Científica em Python
- **Material de aula:** Cap. 4 - Introdução à Programação em Python e GPIO da Raspberry Pi
- Python

Parte 2

Objetivos

O objetivo desta prática é a continuação da familiarização dos conceitos básicos que tange a linguagem de programação Python iniciada na prática anterior, abrangendo os conceitos de GPIO e periféricos da Raspberry Pi para aplicações embarcadas, como PWM, integração com sensores de presença, temperatura e umidade, distância, LEDs, switches etc.

Esta atividade prática consiste em explorar de forma mais ampla as bibliotecas Python GPIO Zero e RPi. GPIO, implementando scripts em Python em sistemas com Linux embarcado por meio do uso da GPIO da Raspberry Pi. Os pinos GPIO (General Purpose Input/Output) da Raspberry Pi podem ser programados como entrada ou saída para realizar várias funções, tais como controle de I/O digital, PWM (Pulse Width Modulation), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), UART (Universal Asynchronous Receiver/Transmitter), entre outras, conforme pinout disponível em: https://pinout.xyz.

As bibliotecas RPi.GPIO e GPIO Zero são usadas para interação com os pinos GPIO, facilitando a programação e o controle de dispositivos e sensores conectados à SBC por meio da linguagem Python. A biblioteca RPi.GPIO oferece uma interface mais direta com os pinos GPIO, ao passo que a GPIO Zero é uma biblioteca de mais alto nível, simplificando todo o controle dos pinos por meio de objetos (ex.: LED.on, LED.off).

A numeração dos pinos ocorre por meio de dois formatos: BCM (Broadcom SoC channel number) e numeração física da placa, conforme pinout. A biblioteca GPIO Zero utiliza o padrão BCM, enquanto na RPi.GPIO é possível definir qual formato por meio de: *GPIO.setmode(GPIO.BCM)* ou *GPIO.setmode(GPIO.BOARD)*. A página de documentação da GPIO Zero disponibiliza códigos base para diferentes aplicações com pinos GPIO por meio das guias “2 -Basic Recipes” e “3-Advanced Recipes” (exemplos: *Traffic lights, Reaction Game, GPIO music box, full color LED/Display, Push Button, Motion Sensor, Servo, Motors, Robot, Multi-character 7-segment Display* etc.). Também é disponibilizado a migração de alguns códigos para a biblioteca RPi.GPIO na guia “10 - Migrating from RPi.GPIO”.

Aplicação

Portanto, a primeira implementação prática será o uso de saídas moduladas por PWM, no qual o fator de trabalho, ou duty cycle, define, dentro do período de um pulso, qual o tempo em que o sinal estará em nível alto ou baixo. Desta forma, em “0” o sinal está sempre desligado, e no máximo valor (“1”) sempre ligado. Em valores intermediários, aumenta-se o tempo em que o sinal está em nível alto, gerando na saída um sinal que possui funcionamento semelhante a um nível constante de tensão intermediário. Sendo assim, deve-se executar um script em Python baseando-se nos exemplos disponibilizados aqui, usando a biblioteca GPIO Zero ou RPi.GPIO, para realizar um controle de um pino na configuração de PWM. Para a visualização do sinal PWM pode-se utilizar o

osciloscópio presente na bancada do Lab.Micros e um LED conectado à placa conforme Figura 1. Variar o duty cycle do sinal PWM e visualizar no osciloscópio. Lembre-se que a conexão do LED deve ser feita respeitando-se os limites de corrente do componente e da Raspberry Pi, conforme segue ilustrado abaixo. :

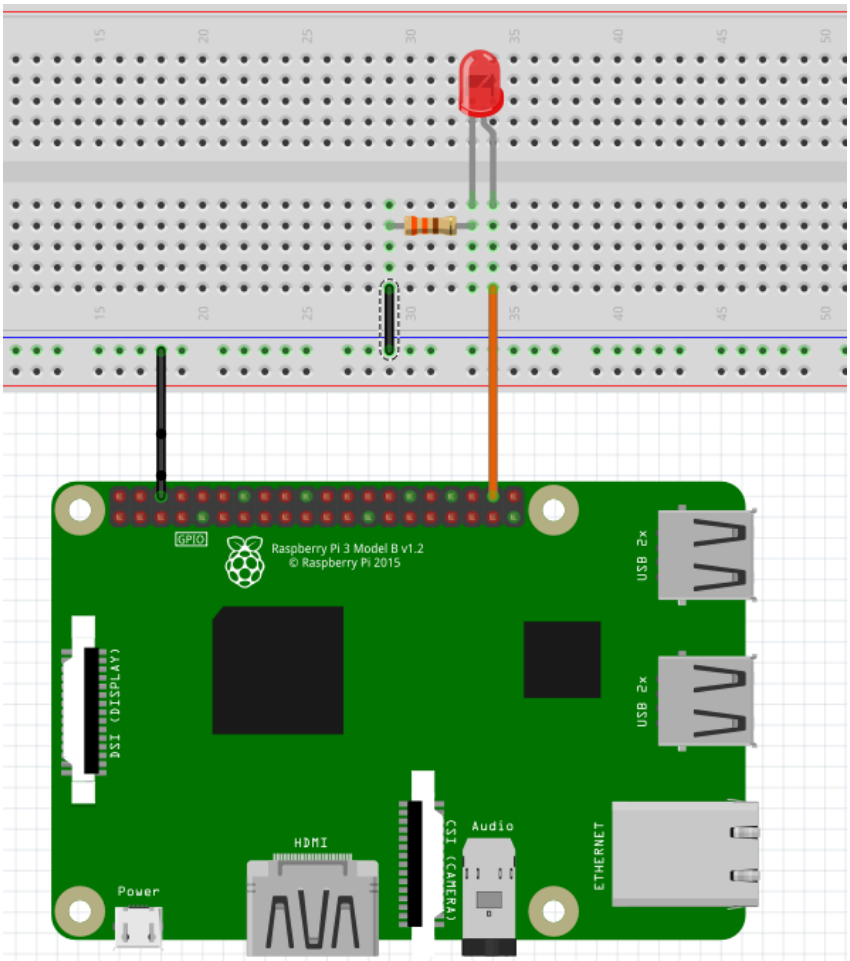


Figura 1 - Montagem para a implementação 2: PWM - O pinout com descrição e numeração de cada um dos pinos pode ser consultado a partir da internet a partir do site de pinout da rasp.

A segunda implementação será de livre escolha, visando explorar a GPIO com uso de sensores/atuadores e as bibliotecas gpiozero e RPi.GPIO. Poderá ser algum exemplo disponibilizado na documentação da GPIO Zero (na guia: 2 -Basic Recipes"), de livre escolha do(a) aluno(a) ou grupo, desde que seja possível implementar na protoboard com componentes físicos existentes no Lab.Micros. Como sugestão, apresenta-se um exemplo de aplicação a seguir (pode ser escolhido algum outro):

> Uso do HC-SR04 (disponível no laboratório), que é um sensor de ultrassom para medir distâncias que opera emitindo um pulso ultrassônico de curta duração e, em seguida, aguarda a reflexão desse pulso em um objeto. Medindo o tempo que leva para o pulso

retornar, o sensor pode calcular a distância até o objeto. Seu alcance varia de 2 cm a 400 cm, dependendo das condições e do ambiente (realizar a montagem prática conforme Figura 3, acrescido da parte de acionamento de um LED conforme montagem prática da Figura 1). Pinos:

- **Vcc**
- **GND**
- **Trig (Trigger)**: conectado a um pino GPIO da Raspberry Pi para enviar o pulso ultrassônico.
- **Echo:** conectado a outro pino GPIO da Raspberry Pi para ler o pulso refletido.

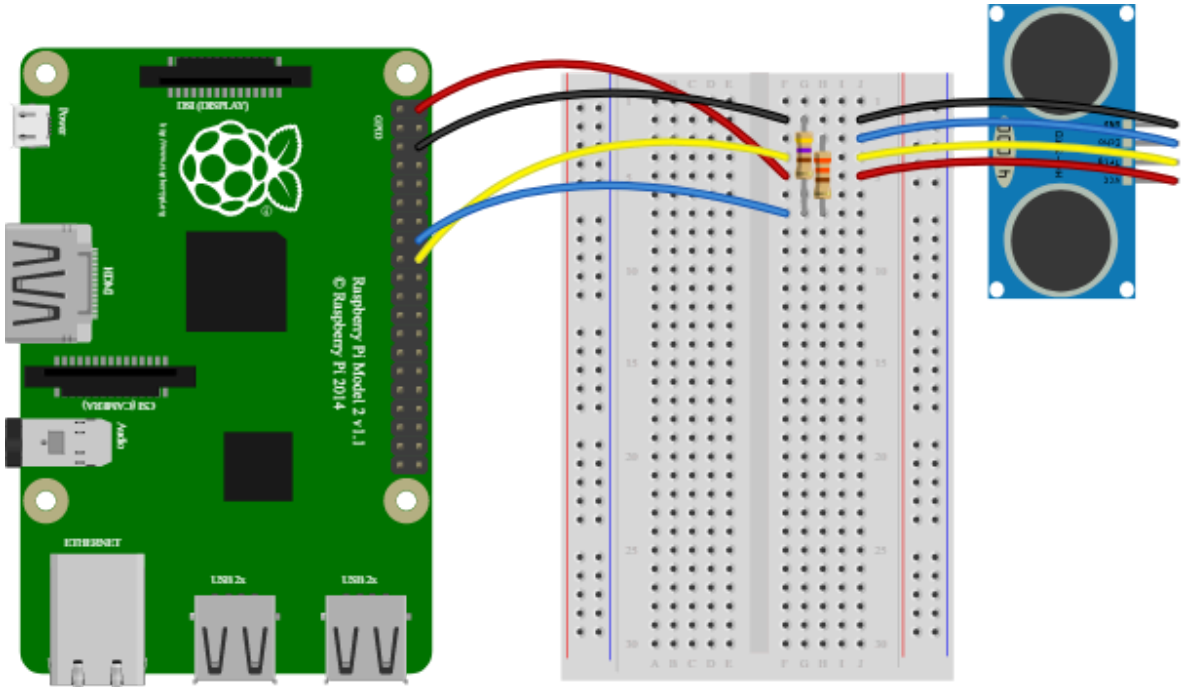


Figura 3 - Sugestão para a segunda implementação: sensor ultrassônico HC-SR04 - para mais detalhes ver documentação da GPIO Zero (item 2 - Basic Recipes) e material de aula (Cap. 4)

Motivação

O uso de PWMs é de extrema importância no contexto de sistemas embarcados, pois permite que os componentes conectados possam funcionar de forma a simular níveis de tensão menores do que os aplicados, sem necessariamente se reduzir a tensão. Esta utilização possui vantagens frente a simples redução da tensão, visto que, em motores de corrente contínua, a redução de tensão provoca redução do torque do motor. Portanto, uma redução da velocidade por amplitude provocaria redução de torque, e em PWM, o torque é mantido. O uso de sensores em sistemas embarcados proporciona automação dos processos e contribuem para melhoria da eficiência e da segurança das aplicações em ambientes residenciais ou industriais. O uso, por exemplo, de sensores de presença permite que dispositivos sejam acionados apenas quando necessário (economizando energia, já que luzes podem ser ligadas e desligadas automaticamente), já que o acionamento é feito somente quando há detecção de movimentos em seu escopo. Da mesma forma, os sensores de distância cumprem um

importante papel na detecção de obstáculos, evitando, por exemplo, colisões em robôs de navegação autônoma. Por fim, destacam-se os sensores de temperatura que operam em sistemas de climatização ou em sistemas de aquecimento, ventilação, e consumo de energia em que se faz necessário ajuste de temperatura com base em condições, sendo imprescindível a coleta de dados climáticos.

Roteiro

- Atente-se à conexão da placa, utilizando os pinos corretos, caso algum dos pinos já esteja em uso na placa, como um ground, utilize outro na placa seguindo a pinagem disponibilizada, e dê preferência a utilizar o ground para conexão do LED.
- Verifique se os estados iniciais dos pinos e do LED estão corretos e não ocasionarão nenhum curto circuito na placa.
- Para a primeira implementação, com PWM:
    - Realizar a montagem prática da Fig. 1
    - Importe a biblioteca RPi.GPIO para uso no projeto.
    - As funções de PWM já estão desenvolvidas nesta biblioteca, veja a documentação ou exemplos no material de aula.
    - Altere a frequência e duty cycle e observe a mudança ocasionada no projeto com o LED e no osciloscópio (será possível observar, sobretudo, a variação da frequência), conectando suas pontas de prova ao circuito montado e ajustando a escala do instrumento.
    - Salve uma foto da visualização do sinal no osciloscópio (pode ser simplesmente uma fotografia tirada com o smartphone).
    - Demonstrar o funcionamento ao professor.

- Para a segunda implementação:
    - Realizar a montagem prática do projeto de sua escolha (conforme sugestões mostradas na Fig. 2 ou outro exemplo na guia “2 - Basic Recipes”, possível de se implementar na prática com os componentes do laboratório).
    - Caso realize a implementação do projeto “Distance Sensor” com sensor ultrassônico HC SR04, o script abaixo mostra um exemplo importando a biblioteca GPIO Zero.
    - Salvar uma foto da montagem prática.
    - Demonstrar o funcionamento ao professor, caso tenha escolhido essa opção.

```
from gpiozero import DistanceSensor, LED
import time
DistanceSensor(echo=24, trigger=23)# pinos da Rasp. escolhidos 24 e 23
While True:
   print("distance is:", sensor.distance, "m")
   time.sleep(1)
```

```
from gpiozero import DistanceSensor, LED from signal import pause
DistanceSensor(echo=24, trigger=23)
led = LED(16)
sensor.when_in_range = led.on sensor.when_out_of_range = led.off
pause()
```

Formato de entrega

- Enviar os scripts em Python ".py" com as linhas de código comentadas referente a cada uma das implementações realizadas na atividade prática, totalizando, portanto, 2 arquivos ".py" (PWM + aplicação escolhida).
- Fazer o upload dos 2 scripts ".py" na tarefa no e-Disciplinas até a data de entrega atribuída.

Critérios de avaliação

- Implementação dos programas e montagem prática dos projetos na protoboard com funcionamento demonstrado e atendimento ao roteiro em termos de requisitos, sequência lógica, uso correto das bibliotecas; participação; trabalho em equipe e boa organização dos experimentos;

Bibliografia

- GPIO Zero
- LOM3260 — Computação Científica em Python
- **Material de aula:** Cap. 4.
- DHT11

Parte 3

Objetivo

O objetivo da terceira parte desta prática é a familiarização do uso de computação paralela, processos e *threads*, concorrência e escalonamento (condições de corrida, multithreading, mutex, semáforos, round-robin e deadlock) em sistemas embarcados com sistema operacional, e mais especificamente Linux embarcado com Raspberry Pi.

Em Python, diferente do que foi abordado para microcontroladores em disciplinas anteriores, não existe um "*timer* interno". Isto se deve ao fato que sistemas operacionais não possuem a responsabilidade com o tempo real. Ademais, aplicações de executando kernel Linux não são adequadas para aplicações críticas de tempo real, já que as funções tempo são organizadas junto às outras entradas da CPU, sua ordem é organizada pelo próprio kernel do sistema e suas bibliotecas, a definir pela prioridade dada a cada atividade.
Porém, pode-se obter sistemas chamados de *soft real time* ou pelo menos a concorrência entre tarefas e processos a partir do uso de *threads* no sistema operacional. Desta forma, como o processador possui mais de um núcleo (no caso da Rasp modelo 3B+, são 4 núcleos), pode-se dedicar a atividade de um núcleo a executar uma tarefa. Assim, os restantes seguem sendo controlados pela ordem de prioridade do sistema, e um deles, porém, é mantido sob prioridade maior do comando passado. Processos e threads advém do conceito de *parallel computing* e CPU *multicore* (Fig. 1).

- **Processos:** qualquer programa rodando em um sistema operacional possui um mais processos associados!
- ***Thread*:** uma unidade de execução de um processo - logo um processo pode consistir de uma ou várias threads (multithreading!).
- **CPU multicore:** possui 2 ou mais núcleos de processamento.

Conceitos

1 - Exemplo de processos

- Controlando uma saída (LED) conectado à Raspberry Pi e, ao mesmo tempo, realizando uma segunda tarefa aleatória (contagem), em processos separados.

```
import RPi.GPIO as GPIO
import time
import random  #para gerar números aleatórios.
from multiprocessing import Process  # classe Process do módulo multiprocessing para criar processos paralelos.
import os  # para interagir com o sistema operacional (obter o PID do processo)

# Configuração da GPIO para piscar o LED
def gpio_blink():
    print(f"Processo de Blink LED iniciado com PID: {os.getpid()}")  # Exibe o PID do processo atual.
    GPIO.setmode(GPIO.BOARD)
```

```
    GPIO.setup(7, GPIO.OUT)
    while True:
        GPIO.output(7, GPIO.HIGH)  # Liga o LED.
        print("LED aceso")
        time.sleep(1)
        GPIO.output(7, GPIO.LOW)  # Desliga o LED.
        print("LED apagado")
        time.sleep(1)
# Função para gerar e exibir uma contagem aleatória
def random_count():
    print(f"Processo de Contagem Aleatória iniciado com PID: {os.getpid()}")  # Exibe o PID do
processo atual.
    while True:
        count = random.randint(1, 100)  # Gera um número aleatório entre 1 e 100.
        print(f"Contagem aleatória: {count}")
        time.sleep(2)  #delay para gerar o próximo número.

if __name__ == '__main__':
    try:
        # Cria dois processos paralelos
        process1 = Process(target=gpio_blink)  # Associa a função gpio_blink ao primeiro
processo.
        process2 = Process(target=random_count)  # Associa a função random_count ao segundo
processo.

        # Inicia os processos
        process1.start()  # Inicia o processo de piscar o LED.
        process2.start()  # Inicia o processo de contagem aleatória.

        # Aguarda a conclusão dos processos (o que não ocorre, pois ambos têm loops infinitos)
        process1.join()  # Espera o término do primeiro processo (infinito).
        process2.join()  # Espera o término do segundo processo (infinito).
    except KeyboardInterrupt:  # interrupção do teclado (Ctrl+C).
        print("Processos interrompidos.")
    finally:
        GPIO.cleanup()  # Limpa a configuração dos pinos GPIO ao encerrar o script.
```

- Executar o script acima no terminal: *python script.py* e abrir um segundo terminal para verificar informações sobre o processo relacionado ao programa pelos comandos abaixo

```
> ps aux | grep python #processos específicos relacionados ao script
> pgrep -fl python #encontrar o ID do processo
> taskset -cp <PID> # afinidade (núcleos que o processo pode rodar)
    taskset -cp 2456 # caso o PID do processo seja 2456
> taskset -cp 2 PID # define a afinidade para núcleo 2
> top -p <PID> # processo em tempo real
> ps aux | grep <PID> ou pgrep -f script.py #mais informações sobre o
processo
  #outros comandos: htop, ps aux, pstreee etc
> python script.py & # o & coloca o processo em segundo plano
> kill <PID> #finaliza o processo
```

2 - Exemplo com threads

- O código abaixo (sem threads) é sequencial, ou seja, executa gpio_blink() e, depois, random_count(). Como as funções têm loops infinitos, o código não sairá dos loops. Ao executar esse programa no terminal, é provável que somente a função gpio_blink seja executada (uma vez que em Python, o código dentro de uma função em loop não permitirá a execução de outras funções simultaneamente se não houver mecanismos para gerenciar a execução paralela, como threads ou processos). Cada função é executada uma após a outra no mesmo fluxo de controle (o código é executado linha após linha). Se a primeira função entra em um loop infinito, o controle nunca alcança a próxima função.

```
import RPi.GPIO as GPIO
import time
import random

# Configuração do GPIO para piscar o LED
def gpio_blink():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(7, GPIO.OUT)  # Usando o pino 7 para o LED

    while True:
        GPIO.output(7, GPIO.HIGH)
        time.sleep(1)  # LED aceso por 1 segundo
        GPIO.output(7, GPIO.LOW)
        time.sleep(1)  # LED apagado por 1 segundo

# Contagem aleatória
def random_count():
    while True:
        count = random.randint(1, 100)
        print(f"Contagem aleatória: {count}")
        time.sleep(2)  # Pausa de 2 segundos entre contagens

if __name__ == '__main__':
    gpio_blink()  # Piscar LED
    random_count()  # Contagem aleatória
```

- A solução é usar threads para executar gpio_blink e random_count simultaneamente! As threads garantem que o programa execute tarefas em paralelo, evitando que uma função com um loop infinito bloqueie a execução das outras. Reproduzir o exemplo abaixo com o interpretador Python e abrir um segundo terminal (enquanto mantém o programa abaixo executando no primeiro terminal) para detalhes sobre as threads/processos em execução (usar os comandos ***ps, top, htop, pstree, e ps -o pid,tid***, no terminal para ver informações).

```
import threading

def primeira_funcao():
    while True:
```

```
        print("Executando a primeira função")

def segunda_funcao():
    while True:
        print("Executando a segunda função")

if __name__ == '__main__':
    # Cria e inicia as threads
    thread1 = threading.Thread(target=primeira_funcao)
    thread2 = threading.Thread(target=segunda_funcao)

    thread1.start()  # Inicia a execução da primeira função em uma nova thread
    thread2.start()  # Inicia a execução da segunda função em outra nova thread

    thread1.join()   # Aguarda a conclusão da primeira thread
    thread2.join()   # Aguarda a conclusão da segunda thread
```

- **Round-robin** é um algoritmo de escalonamento que distribui o tempo de CPU de forma circular entre os processos ou threads. Cada processo ou thread recebe um intervalo de tempo fixo, conhecido como "*quantum*" (quando ele esgota, o próximo processo ou thread na fila é selecionado). No exemplo abaixo, cria-se múltiplos processos que executam por um curto período e monitorar como o sistema operacional alterna entre eles (verificar com ***top*** e ***htop***)

```
import time
import threading

def task(name, duration):
    start_time = time.time()
    while time.time() - start_time < duration:
        print(f"{name} está executando por {time.time() - start_time:.2f} segundos")
        time.sleep(0.5)

def main():
    duration = 5  # Tempo de execução para cada tarefa
    thread1 = threading.Thread(target=task, args=("Thread 1", duration))
    thread2 = threading.Thread(target=task, args=("Thread 2", duration))
    thread3 = threading.Thread(target=task, args=("Thread 3", duration))

    thread1.start()
    thread2.start()
    thread3.start()

    thread1.join()
    thread2.join()
    thread3.join()

if __name__ == "__main__":
    main()
```

- **Preempção:** é um mecanismo que permite que o sistema operacional interrompa um processo ou thread em execução para dar tempo de CPU a outro processo ou thread, garantindo que todos recebam uma fatia justa de tempo do microprocessador. Em sistemas multitarefa, essa ação garante que nenhum processo monopolize a CPU. No exemplo a seguir, cria-se dois processos que fazem uso da CPU. O sistema operacional deve alternar entre esses processos para garantir que ambos recebam tempo do processador (verificar com comandos top e htop)

```
import time
import threading

def cpu_intensive_task(name):
    while True:
        print(f"{name} está consumindo CPU")
        # Simula trabalho intenso de CPU
        _ = sum(x * x for x in range(10000))

def main():
    thread1 = threading.Thread(target=cpu_intensive_task, args=("Thread 1",))
    thread2 = threading.Thread(target=cpu_intensive_task, args=("Thread 2",))

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

if __name__ == "__main__":
    main()
```

- Usando threads para executar gpio_blink e random_count simultaneamente!Tanto o piscar do LED quanto a contagem aleatória devem ser exibidos simultaneamente no terminal. Os prints que mostram o PID do processo principal e o identificador da thread (ps -o pid,tid,comm: exibem o PID e o TID (Thread ID)) (Exemplo: ***ps -eLf | grep <PID>***).

```
import RPi.GPIO as GPIO
import time
import random
import threading
import os

# Configuração do GPIO para piscar o LED
def gpio_blink():
    print(f"Thread para piscar LED iniciada com ID: {threading.get_ident()}")
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(7, GPIO.OUT)  # Usando o pino 7 para o LED

    while True:
        GPIO.output(7, GPIO.HIGH)
        print("LED aceso")  # Mensagem quando o LED está aceso
        time.sleep(1)  # LED aceso por 1 segundo
```

```
        GPIO.output(7, GPIO.LOW)
        print("LED apagado")  # Mensagem quando o LED está apagado
        time.sleep(1)  # LED apagado por 1 segundo

# Contagem aleatória
def random_count():
    print(f"Thread de contagem aleatória iniciada com ID: {threading.get_ident()}")
    while True:
        count = random.randint(1, 100)
        print(f"Contagem aleatória: {count}")
        time.sleep(2)  # Pausa de 2 segundos entre contagens

if __name__ == '__main__':
    try:
        print(f"Processo principal PID: {os.getpid()}")  # Exibe o PID do processo principal

        # Cria threads para executar as funções simultaneamente
        thread1 = threading.Thread(target=gpio_blink)
        thread2 = threading.Thread(target=random_count)

        # Inicia as threads
        thread1.start()
        thread2.start()

        # Aguarda a conclusão das threads
        thread1.join()
        thread2.join()
    except KeyboardInterrupt:
        print("Interrompido pelo usuário.")
    finally:
        GPIO.cleanup()
```

3 - Exemplo com Mutex e Semáforos

- A seguir um exemplo básico (sem mutex) onde várias threads tentam acessar e modificar um recurso compartilhado sem qualquer controle de sincronização. Problema: sem sincronização, o valor final de contagem pode ser incorreto devido a condições de corrida*, onde as threads podem ler e escrever o valor simultaneamente sem coordenação.

```
import threading
import time

contagem = 0

def incrementar():
    global contagem
    for _ in range(100000):
        contagem += 1
    print(f"Contagem final: {contagem}")

if __name__ == '__main__':
    thread1 = threading.Thread(target=incrementar)
    thread2 = threading.Thread(target=incrementar)
```

```
    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    print(f"Contagem final após todas as threads: {contagem}")
```

* Apesar disso, o Global Interpreter Lock (GIL) do Python pode impedir condições de corrida, evitando o problema relatado acima (portanto, pode ser que a contagem aconteça corretamente, pois o interpretador garante uma thread por vez a partir de um mutex interno).

- **Mutex (Mutual Exclusion)** é um mecanismo de sincronização que garante que apenas uma thread possa acessar o recurso compartilhado de cada vez.
- No exemplo abaixo, apenas uma thread de cada vez pode acessar e modificar contagem, evitando condições de corrida

```
import threading
import time

# Recurso compartilhado
contagem = 0
mutex = threading.Lock()  # Cria um mutex

def incrementar():
    global contagem
    for _ in range(100000):
        with mutex:  # Adquire o mutex antes de acessar o recurso compartilhado
            contagem += 1
    print(f"Contagem final: {contagem}")

if __name__ == '__main__':
    thread1 = threading.Thread(target=incrementar)
    thread2 = threading.Thread(target=incrementar)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    print(f"Contagem final após todas as threads: {contagem}")
```

- Usar Mutex ou Semáforo: Mutex garante a exclusão mútua e que apenas uma thread acesse o recurso por vez (quando o recurso é crítico e não deve ser acessado simultaneamente por múltiplas threads).Semáforo permite que um número limitado de threads acesse um recurso ao mesmo tempo

- **Deadlock** ocorre quando duas ou mais threads ficam bloqueadas indefinidamente, esperando por um recurso que é possuído por outra thread
- No exemplo a seguir, Thread 1 adquire Mutex1 e tenta adquirir Mutex2. Thread 2 adquire Mutex2 e tenta adquirir Mutex1. Se Thread1 adquirir Mutex1 e Thread2 adquirir Mutex2 quase ao mesmo tempo, ambas ficarão bloqueadas esperando uma pela outra, resultando em deadlock.

```
import threading
import time

# Recurso compartilhado
mutex1 = threading.Lock()
mutex2 = threading.Lock()

def tarefa1():
    with mutex1:
        print("Thread 1 adquiriu mutex1")
        time.sleep(1)  # Simula algum trabalho
        with mutex2:
            print("Thread 1 adquiriu mutex2")

def tarefa2():
    with mutex2:
        print("Thread 2 adquiriu mutex2")
        time.sleep(1)  # Simula algum trabalho
        with mutex1:
            print("Thread 2 adquiriu mutex1")

if __name__ == '__main__':
    thread1 = threading.Thread(target=tarefa1)
    thread2 = threading.Thread(target=tarefa2)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()
```

- A solução é garantir que todas as threads adquiram os mutexes na mesma ordem. Dessa forma, evitar-se-á o cenário em que uma thread possui um mutex e a outra thread possui o mutex que a primeira precisa.
- No exemplo a seguir, as duas funções (Tarefa1 e Tarefa2) agora adquirem Mutex1 antes de Mutex2, o que evita que uma thread possa adquirir um mutex e a outra thread o mutex que a primeira solicitou, o que evitará o deadlock.

```
import threading
import time

# Recurso compartilhado
mutex1 = threading.Lock()
mutex2 = threading.Lock()
```

```
def tarefa1():
    with mutex1:
        print("Thread 1 adquiriu mutex1")
        time.sleep(1)  # Simula algum trabalho
        with mutex2:
            print("Thread 1 adquiriu mutex2")

def tarefa2():
    with mutex1:
        print("Thread 2 adquiriu mutex1")
        time.sleep(1)  # Simula algum trabalho
        with mutex2:
            print("Thread 2 adquiriu mutex2")

if __name__ == '__main__':
    thread1 = threading.Thread(target=tarefa1)
    thread2 = threading.Thread(target=tarefa2)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()
```

- **Como evitar deadlocks:**
    - Adquirir recursos em uma ordem fixa e hierárquica.
    - Usar timeouts para evitar que uma thread fique bloqueada indefinidamente.
    - Minimizar a necessidade de adquirir múltiplos locks e preferir uma abordagem usando apenas um lock

Atividade Prática para entrega - Parte 3

Portanto, desenvolver uma aplicação prática utilizando os conceitos abordados na parte 3 por meio de um programa em Python e uso da GPIO da Raspberry Pi que faça o uso de processos e/ou threads. Pode-se abordar os programas desenvolvidos no Roteiro da Parte 2 como base ou criar uma nova aplicação nos moldes dos exemplos apresentados anteriormente. O uso de mutex ou semáforos é opcional mas poderá ser aplicado para otimização de um programa. Além disso, pode-se escolher entre usar processos (Multiprocess.Process) ou threads (threading) ou a combinação de ambos.

- **Exemplo de problema:** deve-se realizar uma contagem de tempo, a partir de uma *thread* separada, e após a contagem do tempo, chamar uma função de callback que imprima na tela o resultado do fim da contagem de tempo. Além disso, enquanto a contagem é feita, o código não pode ser mantido em *polling* (quando a CPU ficaria verificando se algum periférico pretende reportar algum evento), pois o código durante a contagem de tempo deve realizar o *blink* de um LED conectado a protoboard, com duas frequências diferentes, e a alteração destas frequências deve ser feita a partir de um botão conectado a Raspberry.

Motivação

O uso de *Threads* em Python é um conceito extremamente importante, não somente para o uso de *timers*, mas para o uso de múltiplas tarefas concorrentes. Dessa forma, não é necessário que se faça múltiplos scripts para funções distintas, além de permitir que os dados não tenham que trafegar entre scripts, o que aumenta a robustez e segurança de sistemas embarcados. Além disso, o conhecimento de ferramentas que permitem que o código possa ser executado sem a parada do código principal é de extrema importância para atividades que exigem responsividade ou que dependam do tempo para serem executadas. É de importância enfatizar que no âmbito de sistemas embarcados, a maioria das aplicações exige que se mantenha alguma atividade sendo executada enquanto se verifica o restante, desde sistemas mais simples, como teclados matriciais em eletrodomésticos (os quais devem sempre estar prontos para leitura de uma entrada do usuário, e não esperar que o usuário clique no botão no momento em que a leitura está sendo feita), até sistemas mais complexos, como máquinas industriais (que não podem interromper seu processo para verificar o acionamento ou entradas de sensores, assim como não podem ficar parada em um delay esperando o fim de um processo, sem que mais nada funcione). Estes são somente alguns exemplos da importância do uso destas ferramentas para o desenvolvimento de sistemas robustos e seguros.

Roteiro

- Atente-se à conexão da placa, utilizando os pinos corretos, caso algum dos pinos já esteja em uso na placa, como um ground, utilize outro seguindo a pinagem disponibilizada.
- Importe as bibliotecas RPi.GPIO, *threading* e/ou *Multiprocess.Process* para uso na aplicação escolhida.
- Implementar e testar com algum periférico disponível no Laboratório (seguindo exemplos dos roteiros das Partes 1 e 2, ou outra aplicação de sua escolha) realizando a montagem prática na protoboard e demonstrar o funcionamento ao professor.
- Sempre verifique se os estados iniciais dos periféricos estão corretos e não ocasionarão nenhum curto circuito na placa quando efetuar as ligações.
- Ao final da execução do programa deve-se atentar a desconectar todas as GPIOs do sistema, a partir do comando em python definido pela biblioteca, evitando que as GPIOs se mantenham em estado ativo após o fechamento do script.

Entrega final

- Enviar o script em Python ".py" com as linhas de código comentadas referente à implementação escolhida para a parte 3.
- Devido a limitação do número de arquivos enviados no e-Disciplinas, pode-se optar por enviar todos os scripts em Python em um único arquivo compactado compreendendo as 3 partes (ou enviar link para o repositório do Github contendo os scripts). Neste caso, o total será de 5 scripts, sendo: GPIO com botão e LED + contagem (Parte 1); PWM + aplicação escolhida na Parte 2; e script da aplicação escolhida na Parte 3.
- Documentação: enviar um documento contendo resumo de no máximo 1 página dos conceitos envolvidos na atividade prática (compreendendo as 3 partes) acrescido de fotos tiradas das montagens práticas realizadas durante as aulas, referentes às 3 partes.

- A documentação acima poderá ser entregue em arquivo PDF. Caso prefira entregar a documentação no GitHub, enviar o link para o repositório onde consta o arquivo “README.md” (pode enviar na tarefa um arquivo txt, com o link dentro, por exemplo - * não enviar diretamente na tarefa o arquivo README.md).

Critérios de avaliação

- Implementação dos programas e montagem prática dos projetos na protoboard com funcionamento demonstrado e atendimento ao roteiro em termos de requisitos, sequência lógica, uso correto das bibliotecas; participação; trabalho em equipe e boa organização dos experimentos;
- Documentação: atendimento ao formato e estrutura solicitada, explicação breve, objetiva e com precisão dos conceitos e passos implementados conforme os roteiros; fotografias das montagens práticas etc.

Bibliografia

An Intro to Threading in Python - Real Python

Material de aula: Cap. 4.
Operating Systems Foundations with Linux on the Raspberry Pi - ARM Education Media.