# SEL0337 - PROJETOS EM SISTEMAS EMBARCADOS


# Prática 5: Configuração do SystemD para Personalização de Serviços de Inicialização de S.O. em Linux Embarcado, Controle de Versão e Repositório de Códigos com Git e GitHub

## Resumo

Essa prática consiste em adicionar uma unidade de serviço personalizada (*systemd service unit*) para gerenciar a inicialização e execução desse projeto em sistemas embarcados com S.O. Linux. Tal ação irá permitir a inicialização automática da aplicação quando da inicialização (boot) do sistema operacional. Como forma de documentação do projeto, o Git será o sistema de versionamento e o GitHub como repositório, junto do histórico de versionamento. Como desafio (opcional), apresenta-se uma proposta de implementação que serve de base para aplicações de reconhecimento facial utilizando interface com a câmera embarcada da Raspberry Pi e machine learning.

## Conceitos importantes:

*Init System, systemd, SysVinit, systemctl, unit file, boot, bootloader, gparted, device tree, device drivers, rootfs*, *Kernel, shell script, bash, Git, GitHub, VCS, version control systems, computer vision, OpenCV, machine learning.*

- **Parte 1 -** Configuração do SystemD para Gerenciar Serviços Personalizados em Sistemas Embarcado
- **Parte 2 -** Git e GitHub em Sistemas Embarcados
- **Parte 3 -** Introdução às Interfaces de Visão Computacional

## Parte 1 - Configuração do SystemD para Gerenciar Serviços Personalizados em Sistemas Embarcados

## Objetivo

Sistemas embarcados de produtos comerciais são específicos e possuem funcionalidades restritas. Aplicações embarcadas, por vezes, necessitam operar já na inicialização do sistema operacional ao invés de aguardar alguém logar no sistema e iniciá-la manualmente. Portanto, essa prática visa explorar o processo de inicialização (boot) do sistema operacional com kernel Linux em sistemas embarcados, utilizando a Raspberry Pi.

Colocar um projeto na inicialização de um sistema operacional embarcado geralmente implica em adicionar uma unidade de serviço personalizada (*systemd service unit*) para gerenciar a inicialização e execução desse projeto. O systemd é um sistema de inicialização moderno adotado em muitas distribuições Linux e sistemas operacionais embarcados.

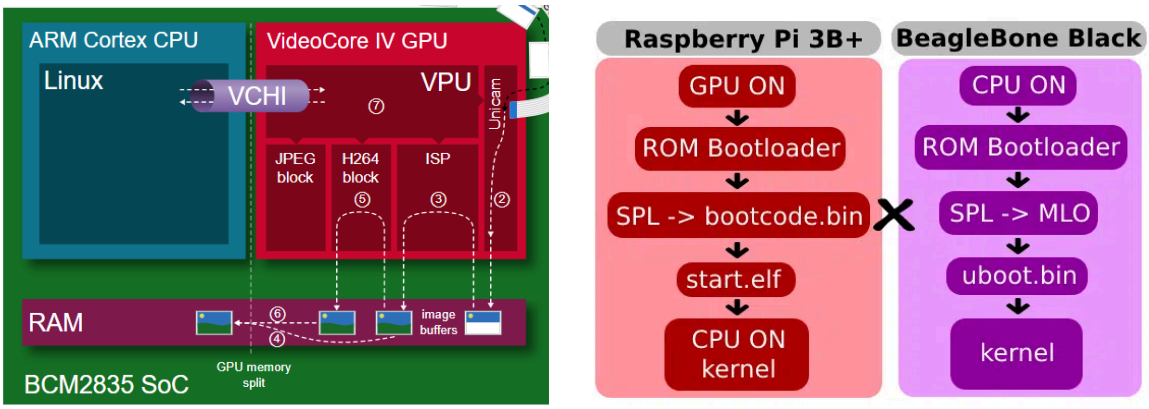
A criação de uma unidade de serviço do *systemd* envolve a criação de um arquivo de configuração que descreve como o projeto deve ser inicializado, parado e gerenciado pelo sistema. Essa unidade de serviço pode especificar coisas como o comando a ser executado, as dependências do serviço, o ambiente de execução, entre outras configurações que serão realizadas na presente atividade prática.

## I - Conhecendo a fundo o "Boot" da Raspberry Pi

No caso da Raspberry Pi, todo o sistema embarcado é comandado pelo SoC (*system on chip*) do fabricante Broadcom (na Rasp.3B+, o modelo é o **BCM2837B0** quad-core, ou seja, com 4 microprocessadores Cortex A-53, microarquitetura ARMv8 64-bit - datasheet aqui). Dessa forma, quando a Rasp. é energizada, o primeiro passo em direção ao *Boot* é ocorre no SoC acima referido.

Ao contrário de outras SBCs, como Beagleboard, por exemplo, na Rasp. quem assume o primeiro controle não é a CPU (ARMv8 - cluster de 4 núcleos responsável por rodar o kernel Linux), mas sim a **GPU** (no caso o VideoCore 3D da Broadcom, responsável pelo processamento gráfico dentro do SoC). Essa ordem é uma característica simplesmente definida pela Broadcom para a família de processadores **BCM283X**.

Fig. 1 - SoC da Rasp. com CPU e GPU; comparativo de Boots entre Rasp. e BeagleBone.




Fonte (imagens): ReadTheDocs

### *First-level bootloader*

Portanto, o primeiro estágio, imediatamente após o *power on*, é denominado "first-level bootloader". Para tanto, um código a ser executado (buscado pela GPU, que inicializa primeiro) é armazenado em ROM dentro do referido SoC, cujo acesso não é disponibilizado pela Broadcom.

> Aliás, cumpre frisar que o detalhamento sobre o primeiro estágio do processo Boot, que ocorre no SoC, é restrito. Apesar da especificação acima, o datasheet disponibilizado pela Broadcom não é completo, detalha apenas o acesso aos periféricos, não descreve o processo

Boot. As informações que temos, portanto, são parciais (o datasheet completo é disponibilizado via NDA- "acordo de não-divulgação")

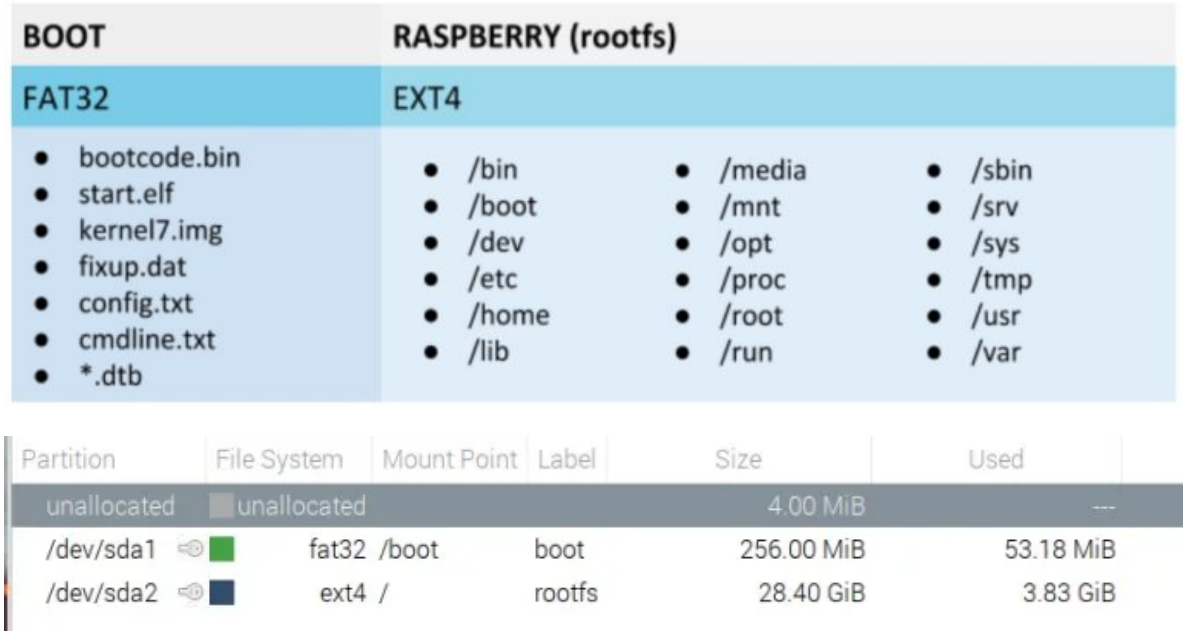
A função do código do *first-level bootloader* é inicializar alguns clocks do processador e buscar por uma memória externa (no caso, o cartão microSD inserido na Rasp - se não houver cartão ele tenta outras fontes: USB e Ethernet). Efetivamente, este primeiro código irá buscar na partição Boot do cartão microSD (primeira partição, do tipo **Fat32 -** *file allocation table*) um segundo código "***bootcode.bin***", para carregá-lo para uma memória interna (cache L2 do SoC) e executar esse arquivo na GPU. Caso desejar visualizar alguns códigos e o sistema de arquivo, execute os comandos abaixo:

```
#Verifique os arquivos em
cd /boot
ls
ls *.bin

#Visualize o particionamento do cartão SD com gparted:
sudo gparted /dev/sd1 # sdX - sd1 ou sd2 ...

# para listagem de discos e mídias:
sudo fdisk -l
lsblk
ls /dev/sdb*
```

Fig. 2 - Particionamento do cartão micro SD com Raspbian instalado.




Fonte (imagem): ARGUS/ RaspberryTips

***Second-level bootloader***

Ocorre quando o arquivo ***bootcode.bin*** é executado, sua função é inicializar a SDRAM (de 1 GB no caso da Rasp. 3B+) para receber outros arquivos de inicialização. Com a SDRAM inicializada, é

também buscado no cartão microSD (na partição boot) um outro arquivo de bootloader chamado ***start.elf*** , o qual também é carregado na memória SDRAM.

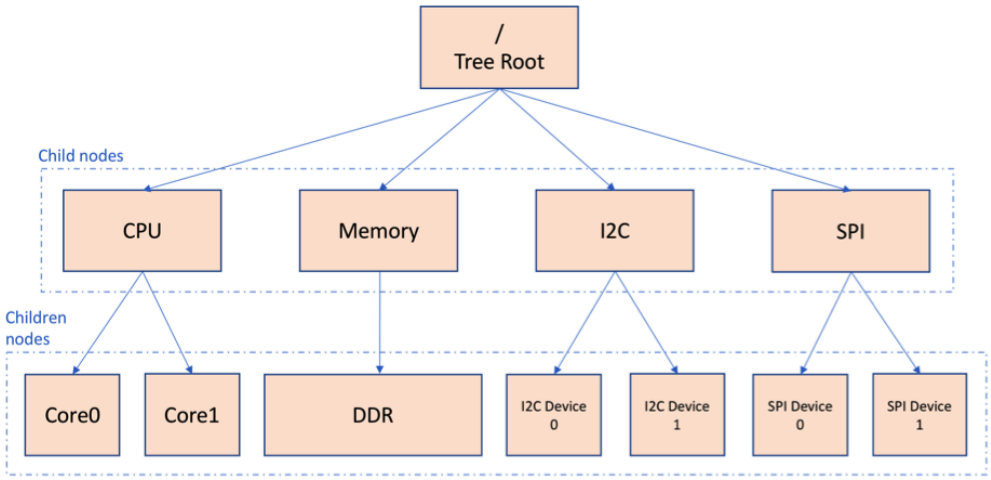
***Third-level bootloader***

O arquivo ***start.elf*** é o firmware da GPU, o qual dá suporte à renderização gráfica. Sua função é buscar os arquivos de device trees (arquivos "*.dtb"*: compilado de representações do hardware que formam o "**device tree"** - Fig. 3) necessários para carregar o kernel Linux. Sua função também (recebendo ajuda de outro binário: ***fixup.dat*** - vide Fig. 1) é configurar o hardware de acordo com os parâmetros que constam em um arquivo de inicialização chamado ***config.txt*** (também localizado na partição *Boot* do cartão microSD).

No arquivo ***config.txt*** constam: configuração do hardware, mapeamento de memória, parâmetros para carregar o kernel Linux.

Por fim, o código ***start.elf*** também busca (sempre na partição boot do cartão microSD) e carrega a imagem do kernel Linux "***kernel.img***" para a memória SDRAM.
A partir daqui, o controle é passado para o kernel Linux conforme parâmetros constantes em um arquivo ***cmdline.txt***, e os núcleos da CPU ARMv8 são inicializados e entram em operação.

Fig. 3- Device tree em sistemas com Linux embarcado.




Fonte (imagem): Octavosystems

***Rootfs* e *init sytem***

Com o kernel Linux em execução, o próximo passo é a inicialização da distribuição Linux (no caso da Rasp. o Raspbian ou Raspberry Pi OS). Aqui, quem entra em operação é o ***init system***. Neste último estágio, são carregados os arquivos da segunda partição do cartão microSD, i.e., a partição "*rootfs*" do tipo "**ext4**" (*extended file system*), responsável pelo sistema de arquivos Linux seguindo formato *root* "/" que conhecemos. O *system**d***, por sua vez, inicializa todos os serviços necessários ao funcionamento da distro Linux, os quais ficam em execução até o momento em que o OS é desligado.

**Resumindo o processo Boot:**

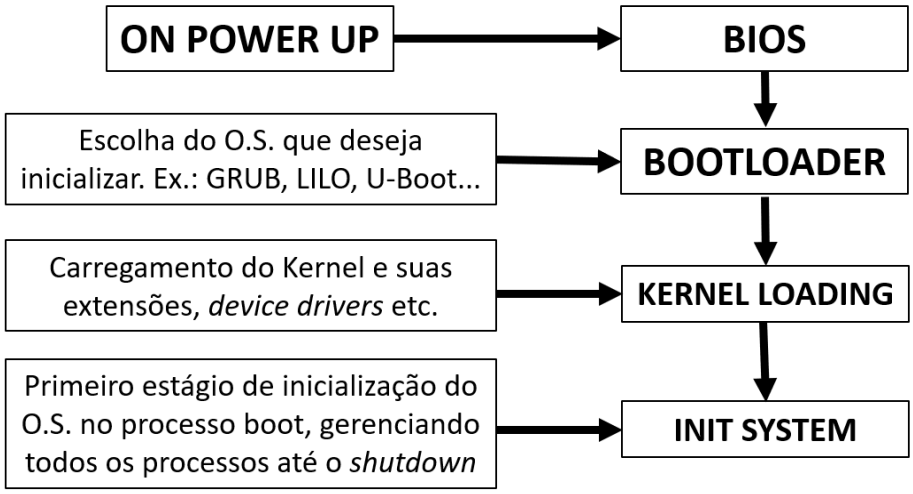
- No *power on*, a GPU é inicializada via código (inacessível) gravado na ROM do SoC BCM enquanto a CPU fica *offline*.

- O primeiro estágio de *bootloader* ocorre quando é buscado, na partição Boot do cartão micro SD com o Raspbian instalado, o arquivo ***bootcode.bin.***
- O segundo estágio de *bootloader* é responsável por habilitar a SDRAM, inicializar periféricos, e buscar pelo arquivo ***start.elf***, que é o firmware da GPU.
- O terceiro estágio de *bootloader* ocorre quando o *firmware* da GPU (arquivo s***tart.elf***) é carregado na RAM e executado na GPU. Neste estágio, a GPU busca a imagem do kernel (***kernel.img***) e também a carrega na RAM. Por fim, ocorre a execução do kernel na CPU ARM.
- Após três estágios de bootloader, temos o estágio ***init sytem***, no qual o kernel monta o sistema de arquivos *root* da segunda partição do cartão micro SD e o *system**d*** executa o processo de inicialização de todos serviços responsáveis pelo funcionamento da distro Linux.
- Todo esse processo Boot é executado em poucos segundos.

## II -*Init System* e *systemd*

O *Init System* é o estágio responsável por todo o processo de inicialização de um sistema operacional (O.S) com o Kernel Linux. Sendo assim, após o computador/SBC ser ligado e dar início ao processo "*boot*", o *bootloader* entra em operação cumprindo a função de carregamento do kernel Linux, conforme detalhado anteriormente. O *Init Sytem*, portanto, é o processo de inicialização do sistema operacional (Debian, Ubuntu, Raspberry Pi OS etc.), que ocorre logo após o Kernel Linux ter sido carregado (Fig. 4).

Fig. 4 - Processo "boot" e etapas de inicialização de um O.S.




### *Systemd*

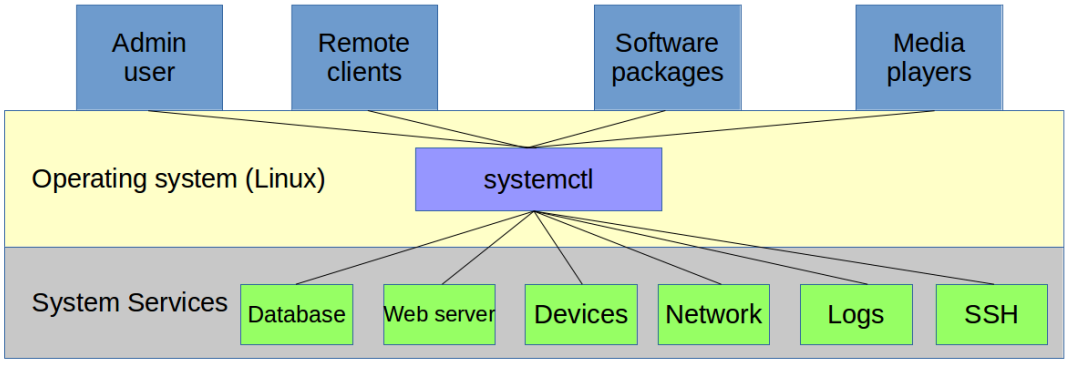
Podemos inferir que o "*system**d***" é uma ferramenta moderna que implementa o estágio "Init System" em uma "distro"(distribuição) Linux (existem outras opções: *system**v***, Upstart, OpenRC). O *system**d*** é, portanto, um conjunto de programas e bibliotecas (é um software livre) responsáveis

pela sequência correta de inicialização e desligamento do sistema operacional (< */etc/init.d*>). É usado em distros modernas, tais como RedHat, Suse, CentOS e Debian. Abaixo, é possível observar o *systemd* na Raspberry Pi, isto é, ao ligar/desligar a placa, note que aparece a tela abaixo com "Ok" na cor verde em cada operação no sistema operacional Raspberry Pi OS (Raspbian), indicando que o referido serviço foi inicializado (ou finalizado) com sucesso (a lista de serviços disponíveis pode ser vista em < */lib/systemd/system*>). Para visualizar a tela abaixo no boot da Rasp, desabilite a opção "S*plash Screen*" em Configurações (*sudo raspi-config > system > splash screen*)

Fig. 5 - Inicialização de serviços com *system**d*** no boot do Raspberry Pi OS


O comando **systemctl** é o gerenciador do *systemd*, i.e., é uma linha de comando do terminal usada para verificar o status dos serviços inicializados ou finalizados pelo *system**d***, operando com um gerenciador utilitário de vários serviços, conforme Fig. 6. O comando **systemctl** também exibe mensagens de erro com maiores detalhes, tais como erros de tempo de execução de serviços, erros de inicialização (< s*udo systemctl status processoX >; < sudo sytemctl start/stop/enable processoX* >).

Fig. 6 - Exemplo de gerenciamento de serviços com *systemd* por meio do comando "systemctl".




***Systemd vs. systemv***

*SysVinit* ou *systemv* ("*system five*" / *Unix System* V/ **SysV**) é o sistema clássico e tradicional de inicialização e gerenciamento de processos em distros Linux, sendo uma das primeiras versões

comerciais do OS Unix. Em uma distro Linux, os scripts de inicialização do ***systemv***, inicializados via shell script, invocam um "*daemon binary*" que irá, então, realizar o "fork" de um processo em background. Embora exista a flexibilidade do *shell script*, a implementação de tarefas é mais complexa no ***systemv*** quando se trata de paralelismo. O novo estilo de ordenação e supervisão de tarefas paralelas do ***systemd***, tornam esse método moderno mais vantajoso em relação ao antigo ***systemv*** (Tabela 1). Ademais, o ***systemd*** também introduziu conceito "***cgroups***" (grupos de controle), que organiza os processos por grupos, seguindo uma hierarquia. Por exemplo, quando processos geram outros processos, os "processos filhos" tornam-se automaticamente membros de um "***cgroup*** pai", evitando confusões sobre a herança dos processos (comparativos: systemd vs. systemv).

Tabela 1 - Comparação: *systemv* vs. *systemd*

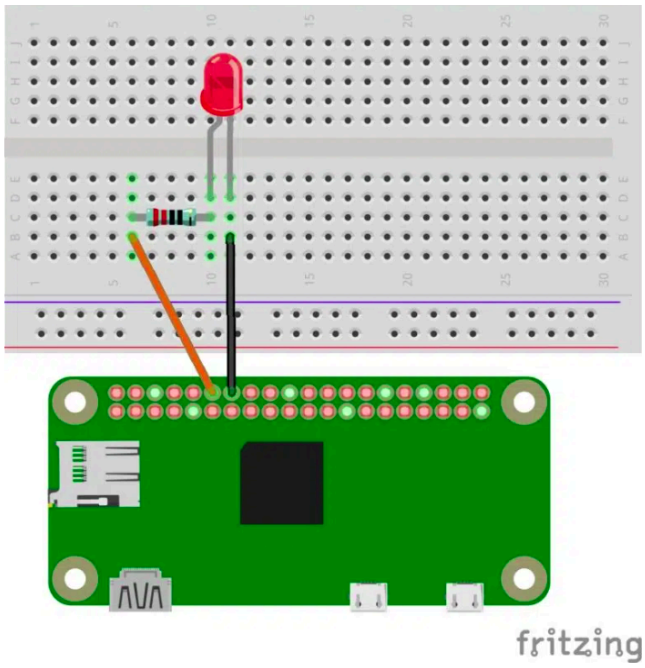
| ***systemv*** | ***systemd*** |
|---|---|
| Mais antigo e popular *init system*, seguindo o padrão Unix | Mais recente e moderno *init system*, usado em várias distros da atualidade (incluindo Raspbian) |
| Inicialização por meio de arquivos de script *shell* | Inicialização via arquivos na extensão "*.service*" |
| API mais antiga | API aprimorada |
| Mais complexo | Design simplificado com maior eficiência |
| Inicialização mais lenta e maior uso de memória e hardware | Inicialização mais rápida utilizando pouca memória |

### Aplicação prática do *systemd* e controle de serviços com *systemctl*

A motivação para uma aplicação prática embarcada utilizando recursos do *systemd* se baseia na necessidade de um aplicativo começar a executar sua função já durante a inicialização do OS, sem ter que aguardar alguém logar no sistema e inicializar o programa manualmente via terminal.

Para a prática, vamos supor que uma aplicação deva inicializar imediatamente após o *power on* da Raspberry Pi. Neste caso, nosso serviço a ser inicializado no processo boot será um script que realiza o *blink* de um LED. Primeiramente devemos criar um script segundo a montagem da Fig. 7, para acesso a GPIO da Rasp. Por fim, será necessário colocar esse serviço à disposição do *systemd*, por meio de um unit file, para que o LED comece a piscar já na inicialização do Raspberry Pi OS.

Fig. 7. Montagem prática do circuito que será usado como serviço no *systemd*




**Roteiro (Parte 1):**

- Caso ainda não tenha sido feito, execute a seguinte atualização no terminal Linux da Raspberry Pi: sudo rpi-update cac01bed1224743104cb2a4103605f269f207b1a #6.1.54
- Sugere-se copiar e colar para otimizar o processo. A atualização acima deve ser feita para garantir compatibilidade de bibliotecas do S.O. para que os exemplos abaixo funcionem. Outra opção seria implementar seguindo a estrutura das novas bibliotecas.
- Faça a ligação física do circuito ilustrado na Fig. 7. e em seguida ligue a Raspberry Pi.
- Crie um diretório de trabalho (não será necessário usar ambiente virtual) e escreva um programa para realizar o *blink* de um LED conectado à GPIO da Rasp. Desta vez, vamos fazê-lo em bash script, para aprendermos a manipular o acesso direto à GPIO via sistema.
- No script, é possível observar como ocorre caminho de acesso direto à GPIO, auxiliando na compreensão de como o sistema de arquivos Linux é montado (rootfs).

```
#!/bin/bash

echo 18 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio18/direction

while [ 1 ]
    do
        echo 1 > /sys/class/gpio/gpio18/value
        sleep 0.2s
        echo 0 > /sys/class/gpio/gpio18/value
        sleep 0.2s
    done
```

- **#!/bin/bash:** Esta linha indica que o script deve ser executado usando o interpretador Bash (#!)
- **echo 18 > /sys/class/gpio/export:** exporta o pino GPIO 18 (neste exemplo foi usado o 18, mas pode ser outro pino), tornando-o disponível para controle. O sistema de arquivos */sys/class/gpio* é usado para interagir com os pinos GPIO.
- **echo out > /sys/class/gpio/gpio18/direction:** configura o pino GPIO 18 como uma saída (direction é usado para definir a direção do pino).
- **do:** início do bloco de comandos que serão executados dentro do loop.
- **echo X > /sys/class/gpio/gpio18/value**: define o valor do pino GPIO 18 como 1, que corresponde a nível alto (liga o LED) ou como 0 - nível baixo (desliga o LED).
- **done:** fim do bloco de comandos que estão dentro do loop.

- Salve o programa como *blink.sh*, que será o serviço que deverá entrar em operação na inicialização da Raspberry Pi.
- Em seguida, é necessário conferir a permissão de execução ("**x"**) neste arquivo, utilizando o comando "*chmod*", responsável por alterar permissões ("+" significa "adicionar" permissão)

```
chmod +x blink.sh
```

- Execute o arquivo para checar se está funcionando. Se estiver Ok, o script irá fazer com que o LED pisque infinitamente. Digite "*ctrol+c*" para encerrar.

```
./blink.sh

# Se desejar, verifique o caminho de acesso a GPIO: ls /sys/class/gpio
```

- O próximo passo é a criação de um ***unit file***, i.e, um arquivo responsável por colocar o serviço que criamos à disposição do *systemd*.
- Portanto, criar um arquivo chamado "*blink.service*", com o conteúdo abaixo. Lembrando que o systemd executa serviços na extensão ".service".

```
[Unit]
Description= Blink LED
After=multi-user.target

[Service]
ExecStart=/home/sel/blink.sh
#ExecStop=
user=SEL

[Install]
```

```
WantedBy=multi-user.target
```

- Observa-se que o arquivo é composto por três principais blocos:
    - **[Unit]** - primeira seção responsável pelas informações do serviço e descrição de suas dependências. O comando ***Description*** entrega essa informação, que será escrita na tela de inicialização do systemd, quando da inicialização da Rasp., com "**OK**" na cor verde se o serviço foi inicializado com sucesso ou "**Failed**" em vermelho.
    - **[Service]** - configurações da execução do serviço que será inicializado: ***Type*** (forma como os processos/scripts serão executados); ***ExecStart*** (arquivo que será executado na inicialização); e ***ExecStop*** (arquivo que será executado ao parar o serviço). Note que foi dado "*export*" na GPIO 18, conforme consta no script *blink.sh*. Logo, seria possível a escrita de um outro script que poderia efetuar o "*unexport*" da GPIO 18 e, portanto, parando o serviço, informando isso em ***ExecStop***.
    - **[Install]** - descreve o comportamento da inicialização do serviço. O ***WantedBy*** informa ao *system**d*** o grupo alvo no qual o serviço que deverá ser inicializado faz parte (exemplo: grupo multi-user).

- Sabemos agora, portanto, que o system**d** é composto por vários destes *unit files* que especificam quais serviços (verifique com *ls /lib/systemd/system*), como devem ser inicializados, e quais são as suas dependências para operar. Em nosso exemplo, passamos esses parâmetros ao system**d**, via *unit file* "*blink.service*", para inicializar o *blink.sh.*
- Copie o arquivo "*blink.service*" para o diretório do padrão de serviços do *system**d***, para que de fato seja reconhecido e esteja à disposição no estágio *init system*.

```
sudo cp blink.service /lib/systemd/system/
```

- Em seguida, inicialize o utilitário ***systemctl*** para testar o serviço:

```
sudo systemctl start blink
```

- Neste caso, o ***systemctl start*** faz com que o *system**d*** execute o serviço informado em **ExecStart** no unit file "*blink.service*", que em nosso caso é o script "*blink.sh*".
- Observe também que não é necessário informar a extensão, pois o ***systemctl*** compreende que o arquivo passado é o *blink.service*, pois estruturalmente, o *system**d*** executa arquivos com essa extensão.
- Para parar o serviço, utilize também o utilitário ***systemctl***:

```
sudo systemctl stop blink
```

    - Opcional: após o stop, poderia ser executado um outro programa, caso fosse informado em ExecStop dentro do arquivo "*blink.service*" anteriormente criado. Por exemplo, poderia ser criado um programa, assim como o blink.sh que pisca o LED, mas que ao invés disso, poderia dar um unexport e desligar o LED. Esse novo programa deve ser informado no arquivo blink.service adicionando a linha: `ExecStop=/home/sel/blinkoff.sh` (em que "blinkoff.sh" é o nome do

programa). Assim, esse programa seria executado toda vez que o serviço for parado pela linha *sudo systemctl stop blink*.

- O próximo e último passo é a habilitação do serviço durante o *Boot* do OS, i.e., até o passo anterior, o serviço que criamos está funcional e à disposição do *system**d***. No entanto, irá funcionar somente ao executar ***systemctl start***.
- Portanto, habilite o serviço para que o LED comece a piscar já na inicialização da Rasp. usando a função "***enable***".

```
sudo systemctl enable blink
```

- Se estiver tudo Ok, o serviço deve inicializar automaticamente durante o estágio *init system* no *Boot* da Raspberry Pi.
- Reinicie a Raspberry Pi para testar a funcionalidade criada. Se necessário, desabilite a opção "S*plash Screen*" em Configurações da Rasp. (sudo raspi-config - system - splash screen) para visualizar as mensagens de inicialização. A opção *splash screen*, quando habilitada, não mostra esses detalhes.
- O serviço deve ser inicializado no estágio *init system* e o LED deve começar a piscar. O serviço é identificado pela mensagem que foi digitada em ***Description***, no *unit file* "*blink.service*", com um "OK" verde na frente, conforme resultado abaixo.

```
[    OK    ]  Started  Blink LED . .
```

- Para solução de problemas, utilize ***systemctl status*** e verifique mensagens de erro.

```
systemctl status blink.service
```

- Para desabilitar o serviço no Boot, basta utilizar:

```
sudo systemctl disable blink
```

- Verifique com ***systemctl statuts***. Para recarregar o serviço após alterações feitas utilizar a linha abaixo. Vale lembrar que o programa blink.service deve ser novamente copiado para o caminho */lib/systemd/system* caso seja alterado (ou alterá-lo diretamente neste local)

```
sudo systemctl daemon-reload
```

**Tarefa (Parte 1):**

Repita o processo acima, criando e testando um novo serviço que deve ser inicializado no Boot da Raspberry Pi pelo *systemd*. No entanto, ao invés do arquivo bash .sh, deve ser inicializado um programa em Python a ser chamado pelo *unit file*. Portanto, escolha um programa em Python já desenvolvido nas práticas anteriores (pode ser um blink LED ou outro) e coloque o projeto no *systemd* implementando os passos acima. Observe que neste caso o próprio Python também é um serviço que deve ser chamado/inicializado no *unit file*, pois o *script.py* depende dele para execução em razão de suas bibliotecas. Adicione também no *unit file* um programa qualquer que deve ser executado quando o serviço for pausado (em ExecStop =) com *sudo systemctl stop programa*.

Realize a montagem prática do projeto escolhido e teste o funcionamento verificando sua inicialização quando do boot da Raspberry Pi. Demonstrar o funcionamento ao professor.

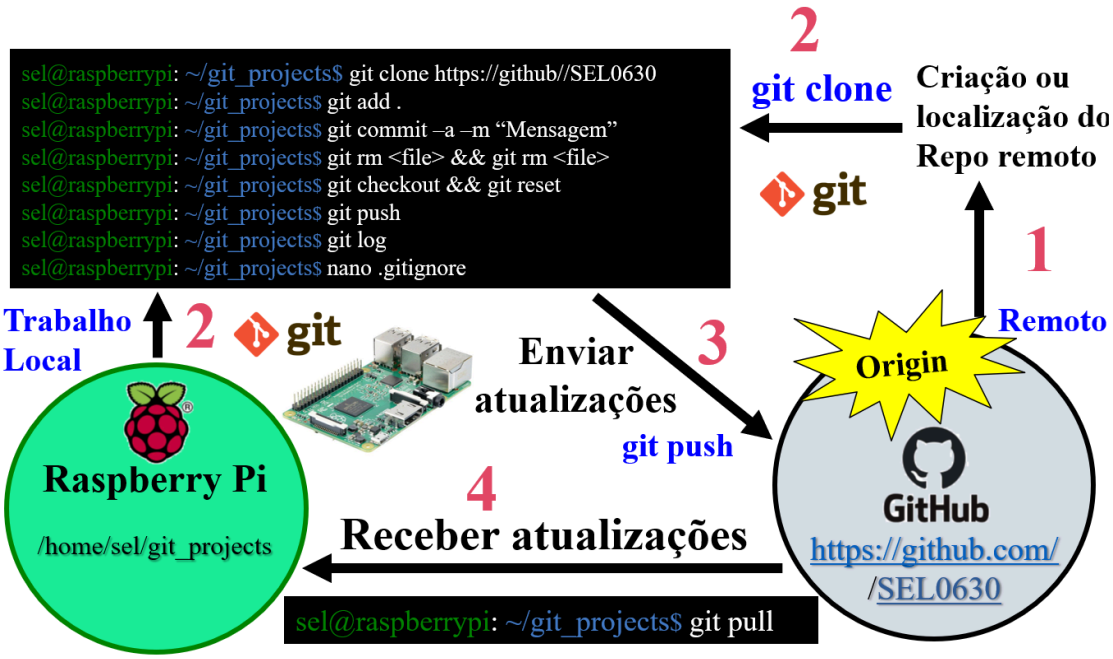
**Bibliografia**


Systemd
Systemd *vs* SystemV
Systemctl


## Parte 2 - Git e GitHub em Sistemas Embarcados

O uso do sistema de versionamento de código Git e o sistema de armazenamento destas versões e documentação GitHub é de extrema importância para os projetos profissionais desenvolvidos por empresas e grupos de software livre. Além disso, serve como forma de distribuição de softwares, e pode ser usado para projetos em que possua um número grande de pessoas e corporações atuando em um mesmo projeto, uma vez que o sistema é capaz de cuidar das inconsistências, e reunir alterações no código feitas em partes distintas do projeto, sem que uma parte interfira na outra. Os repositórios em GitHub também servem como forma de documentar projetos grandes, a fim de garantir que outras pessoas possam implementar a sua solução em outros projetos, garantindo que o seu projeto possa ajudar mais pessoas, e seja encontrado de forma fácil.

Fig. 8- Ideia do projeto com documentação usando Git/GitHub



**Roteiro (Parte 2):**

- Configure o Git na Raspberry Pi, e realize o controle de versão dos códigos gerados nesta prática (explore os comandos: `git init`, `git clone https://…`, `git add`, `git commit -m "mensagem"`, `git push`, `git pull`, `git reset`, `git checkout`, `git rm`, `git ignore`, e `git log`).
- Criar uma conta no GitHub e um repositório com código da disciplina e documentar o projeto dentro dele a partir da Raspberry Pi.
- Realizar um clone do repositório na Raspberry Pi, inserir os programas em python desenvolvidos, e realizar commits e *push* para enviar as atualizações que realizar localmente para o repositório remoto na sua conta do GitHub.
- Configurar seu acesso por meio dos comandos `git config --global user.name "username" #usuário do GitHub entre aspas` e `git config --global user.email "seu_email" #entre aspas`
- Lembre-se de implementar todas as etapas utilizando os comandos em git aprendidos, de forma a manter um histórico das suas versões de código, e documentadas as suas alterações.
- Para realizar o push é necessário, na sua conta do GitHub, cadastrar um Token (salve a chave em um arquivo de texto e deixe disponível na Raspberry Pi para ser copiada quando for necessário fazer um push). Ver Tutorial ao final do roteiro.
- Quando for solicitado um login para realizar um push, deve ser informado o usuário da sua conta do GitHub e, no lugar da senha, deve ser informado o Token gerado conforme instruções no tutorial abaixo (ver Tutorial para geração de uma chave de segurança na sua conta do GitHub).
- Gere um arquivo README.md no repositório, e detalhe o funcionamento básico dos projetos de forma resumida da Prática 5 (Parte 1), adicionando também as fotografias/prints tirados nesta prática (ver seção "Entrega" para mais detalhes sobre o que será avaliado).
- No terminal do repositório local na Raspberry Pi, execute por fim o comando `git log` para ver o histórico de commits, mensagens, e modificações realizadas nas versões dos códigos.
- Salve o log gerado pelo comando anterior, direcionando para um arquivo txt: `git log > historico_git.txt`

**Entregas (Parte 1 e Parte 2) - documentação no GitHub**

- Link para o arquivo README.md no repositório do GitHub com resumo do funcionamento do projeto (Parte 1) e fotografias da montagem realizada no laboratório (uma fotografia da montagem prática do projeto/print do funcionamento do programa solicitado no item Tarefa da Parte 1).
    - **OBS.** Para esta prática, não haverá entrega de relatório em outros formatos (como PDF, por exemplo). Uma vez que a prática versa sobre uso do Git/GitHub para controle de versão e documentação, **é obrigatório a entrega da documentação exclusivamente por meio do README.md**. Neste caso, pode ser enviado na tarefa um arquivo .txt com o link para o repositório onde consta esse arquivo (**não enviar diretamente na tarefa o arquivo .md**).
- Arquivo "txt" contendo salvo o log gerado pelo Git no repositório local, na Raspberry Pi, após ter digitado `git log`.

- Scripts em python ".py" e *unit file* ".service" desenvolvidos na tarefa, contendo, de forma comentada nos próprios scripts, uma explicação resumida das linhas de código e dos conceitos desta prática. **Estes scripts podem ser enviados no repositório do GitHub em questão.**
- Caso realize essa prática de forma antecipada e fora do horário de aula, enviar também um vídeo curto demonstrando o funcionamento do projeto (mostrar a montagem prática inicializando seu funcionamento a partir de quando ocorre o boot da Raspberry Pi), que irá substituir a demonstração do funcionamento do programa ao professor.

**Tutorial para geração de Token na conta do GitHub:**

- Acesse a aba "Settings" na sua conta e a guia "Developer Settings"
- Em seguida, acesse a aba "Personal Access Tokens"
- Em seguida, escolha a opção: "generate a new token" – Classic.
- Defina um nome.
- Em "Expiration" escolha o prazo mais curto;
- Em "Select Scope" habilite todas as opções
- Copie a chave gerada para algum local no computador ou salve ela em arquivo txt.
- **É importante realizar o passo acima, pois após fechar a janela de geração do Token, por questões de segurança o GitHub não permitirá que você veja novamente o Token gerado (terá que criar outro caso não tenha copiado/salvo em algum lugar).**
- Além disso, será importante copiar o Token gerado para um arquivo e deixá-lo à disposição no mesmo diretório que estará trabalhando na Raspberry Pi.
- Ocorre que ao executar, no repositório local na Raspberry Pi, o comando git push (comando para enviar modificações para o repositório remoto correspondente, na sua conta do GitHub), será solicitado inserir seu usuário e senha da sua conta no GitHub, onde foi criado o repositório remoto. Entretanto, **no lugar da senha da sua conta, deverá ser informado o Token que foi gerado anteriormente, para que de fato tudo funcione corretamente quando executar git push** (é uma medida de segurança do GitHub). Por essa razão, será mais fácil manter essa chave salva em um arquivo local, na Raspberry Pi, de forma que possa copiar a chave (usando o comando cat para mostrar o conteúdo do arquivo, no qual salvou o Token) antes de executar o comando git push.

# Parte 3 - Desafio (Opcional * pontuação diferenciada)

## Introdução às interfaces de visão computacional

**Resumo**

Introdução ao uso de periféricos embarcados na Raspberry Pi com o uso do módulo de câmera como base para projetos de reconhecimento facial em banco de dados e servidores.

**Conceitos importantes**

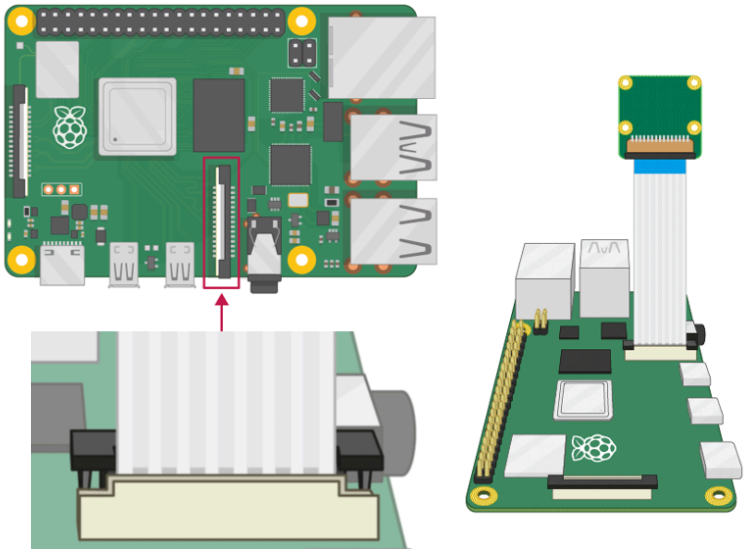
OpenCV, PiCamera, visão computacional, banco de dados, web server, machine learning.

**Objetivo**

O objetivo desta prática é desenvolver um projeto que sirva de base para para reconhecimento facial em sistemas embarcados. Para isto, será utilizado o módulo da câmera embarcado na Raspberry Pi.
É recomendado o uso da biblioteca OpenCV, amplamente usada em visão computacional, e do método Haar Cascade, o qual é um algoritmo de *machine learning* (aprendizado de máquina) que opera como um classificador, buscando por padrões faciais analisando a intensidade dos pixels a partir de uma base de dados. Outras estruturas e bibliotecas, bem variações da aplicação aqui sugerida podem ser implantadas a critério do grupo de forma que seja obtido resultado equivalente.

**Aplicação**

Portanto, para a prática, deve-se executar um script em Python, que inicialmente seja capaz de obter dados da câmera, seguindo roteiro e tutorial disponibilizado a seguir. A partir disso, deve-se implementar alguma modificação na imagem. Utilizar um preview de imagem para verificar a posição da câmera, e a realização de uma foto usando o algoritmo inteligente Haar Cascade para detecção facial e desenhando um retângulo que delimita a face detectada (ver tutorial).



Posição da câmera na conexão via cabo flat

**Motivação**

*Machine learning* é a capacidade dos computadores de reconhecer padrões, não apenas a sistemas com câmeras, mas também a filtragem de emails, sistemas de verificação e chatbots. A tecnologia aprende com dados para tomar decisões e oferecer funcionalidades, sendo muito utilizada em sistemas embarcados. O método de classificação de imagens Haar Cascade, proposto por Viola e Jones em 2001, envolve treinar um modelo para reconhecer padrões em imagens. Esse modelo é treinado com imagens positivas (contendo o objeto a ser reconhecido) e negativas (sem o objeto). O algoritmo analisa a intensidade dos pixels e busca padrões em cascata, otimizando o processo. Neste caso, a biblioteca OpenCV fornece modelos pré-treinados para detecção de faces, olhos, sorrisos, etc., facilitando seu uso. Os modelos são arquivos XML (linguagem de marcação com tags para organizar e estruturar informações) com coordenadas para verificar características nas imagens de entrada. Neste contexto, as bibliotecas de visão computacional capacitam o processamento de imagens e análise de vídeo para aplicações de reconhecimento facial, permitindo a autenticação baseada em identidade biométrica, contribuindo para a segurança.

**Roteiro (Parte 3)**

- Atente-se à conexão da placa: a câmera deve ser conectada enquanto a placa está desligada. No slot que está escrito câmera, entre o conector ethernet e o HDMI da placa, verifique se a posição dos conectores do cabo flat e dos pinos internos do conector estão de acordo, evitando curtos circuitos na câmera (tutorial de como conectar: https://abrir.link/mY6WK).
- Para testar o funcionamento da câmera, pode-se utilizar o comando do terminal *libcamera-hello* (ver outros comandos na documentação da libcamera da Raspberry Pi) (**verificar se houve atualização da documentação e da versão da biblioteca**).
- Fazer o download do algoritmo Haar Cascade por meio do arquivo **haarcascade_frontalface_default.xml** disponível aqui

- Instale as bibliotecas Python OpenCV e PiCamera2: pip install opencv-python e pip install picamera2 (verificar se é necessário usar uma estrutura alternativa ou se houve atualização nos comandos)
- Importe os módulos "*PiCamera2", e "cv2"* (OpenCV) e implemente o script fornecido como exemplo ao final do roteiro. Trata-se de um projeto para detecção facial. Teste o programa, verificando se reconhece seu rosto.
- Este programa, após detectar um rosto, tira várias fotos, as quais são salvas dentro de uma pasta criada ao executar o script ("output_directory").
- Lembre-se que dentro do programa deve ser informado o caminho no qual encontra-se o arquivo **haarcascade_frontalface_default.xml**, localizado na sua Raspberry Pi, após ter feito o download dele.
- A partir deste programa de exemplo (estudar o código), implementar alguma modificação na imagem (pode ser envolvendo a GPIO: um botão acionado tira uma foto; ou um LED é acionado quando a foto é tirada ou reconhecida, por exemplo, ou alguma outra inserção que desejar), e/ou escrever na própria imagem o n° USP da dupla que está executando o projeto.
- Demonstrar o funcionamento por meio da gravação de um vídeo.
- Tirar uma fotografia da montagem prática e um print/fotografia da tela da Raspberry Pi que demonstre o programa funcionando (ou usar uma das fotos salvas na "output_directory" tiradas pela própria câmera da Rasp. quando da execução do programa).

**OBS:** o objetivo aqui é que esta implementação sirva de base para projetos maiores e mais específicos à uma situação prática de sistemas embarcados. Portanto, sinta-se à vontade para implementar modificações que não foram solicitadas no roteiro e para usar a criatividade para agregar outras funcionalidades relativas à visão computacional, machine learning e IA.

**Entrega (Parte 3 - desafio - opcional)**

- O script completo desenvolvido em Python ".py" para o projeto da câmera com alguma modificação agregada.
- Documentação em arquivo PDF ou link para o arquivo README.md no repositório do GitHub com resumo do funcionamento do projeto e fotografias da montagem realizada no laboratório.
- A Parte 3 poderá ser integrada à documentação das Partes 1 e 2 diretamente no GitHub, constituindo uma única entrega.

**Bibliografia**


- Raspberry Pi - Libcamera Documentation
- Biblioteca PiCamera2 com Python
- Biblioteca OpenCV na Raspberry Pi

**Exemplos de uso das bibliotecas Python OpenCV e PiCamera2 para detecção facial usando a câmera da Raspberry Pi (Parte 3 - desafio - opcional)**

```
#!/usr/bin/python3

import cv2 # Biblioteca OpenCV
import os  # Biblioteca para operações do sistema
import time  # Biblioteca de tempo
from picamera2 import Picamera2  # Biblioteca da câmera da Raspberry Pi

# Carrega o classificador para detecção facial (informar o caminho do arquivo)
face_detector = cv2.CascadeClassifier("/home/sel/haarcascade_frontalface_default.xml")

# Inicia uma thread para gerenciar janelas de visualização
cv2.startWindowThread()

# Inicializa a câmera da Raspberry Pi
picam2 = Picamera2()

# Configura a câmera para criar uma visualização com formato de representação
de cores 32 bits “XRGB8888” e resolução de 640x480 pixels
picam2.configure(picam2.create_preview_configuration(main={"format":
'XRGB8888', "size": (640, 480)})

# Inicia a câmera
picam2.start()

# Define o diretório onde as imagens com rostos detectados serão armazenadas
output_directory = "detected_faces"

# Cria o diretório, se ele não existir
os.makedirs(output_directory, exist_ok=True)

# Loop para captura e detecção de rostos
while True:
    # Captura um quadro da câmera e armazena na variável
    im = picam2.capture_array()

    # Converte a imagem colorida para escala de cinza
    grey = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    # Usa o classificador em cascata para detectar rostos na imagem em escala
de cinza
    faces = face_detector.detectMultiScale(grey, 1.1, 5)

    # Loop para processar cada rosto detectado
    for (x, y, w, h) in faces:
```

```
        # Desenha um retângulo verde ao redor do rosto na imagem original
        cv2.rectangle(im, (x, y), (x + w, y + h), (0, 255, 0))

        # Gera um nome de arquivo único com base no carimbo de data/hora
        timestamp = int(time.time())
        filename = os.path.join(output_directory, f"face_{timestamp}.jpg")

        # Salva apenas a porção da imagem que contém o rosto detectado como um
arquivo JPEG
        cv2.imwrite(filename, im[y:y+h, x:x+w])

    # Exibe a imagem com os retângulos desenhados em uma janela com o título
"Camera"
    cv2.imshow("Camera", im)

    # Aguarda 1 milissegundo antes de continuar o loop e capturar a próxima
imagem
    cv2.waitKey(1)
```