



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.

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.

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**.

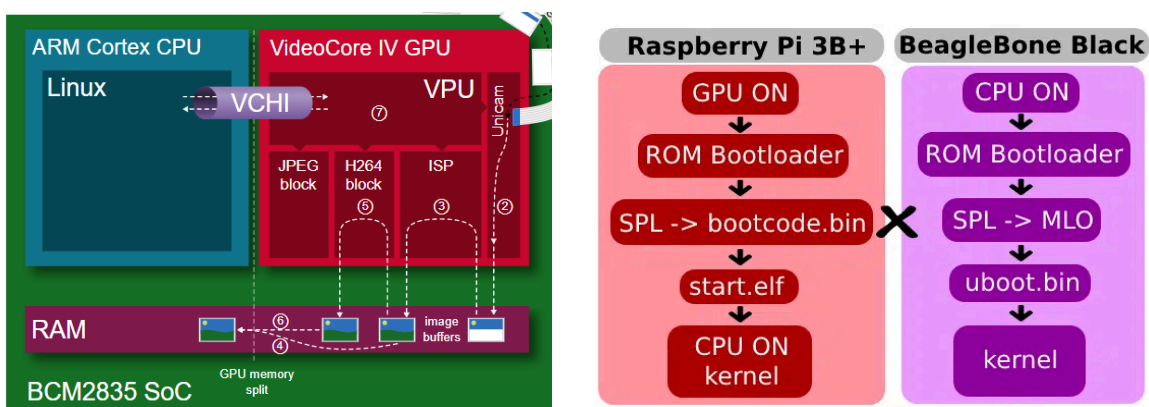


Fig. 1 - SoC da Rasp. com CPU e GPU; comparativo de Boots entre Rasp. e BeagleBone.

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*
```

BOOT

FAT32

- bootcode.bin
- start.elf
- kernel7.img
- fixup.dat
- config.txt
- cmdline.txt
- *.dtb

RASPBERRY (rootfs)

EXT4

- /bin
- /boot
- /dev
- /etc
- /home
- /lib
- /media
- /mnt
- /opt
- /proc
- /root
- /run
- /sbin
- /srv
- /sys
- /tmp
- /usr
- /var




Partition	File System	Mount Point	Label	Size	Used
unallocated	 unallocated			4.00 MiB	---
/dev/sda1	 fat32	/boot	boot	256.00 MiB	53.18 MiB
/dev/sda2	 ext4	/	rootfs	28.40 GiB	3.83 GiB

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.

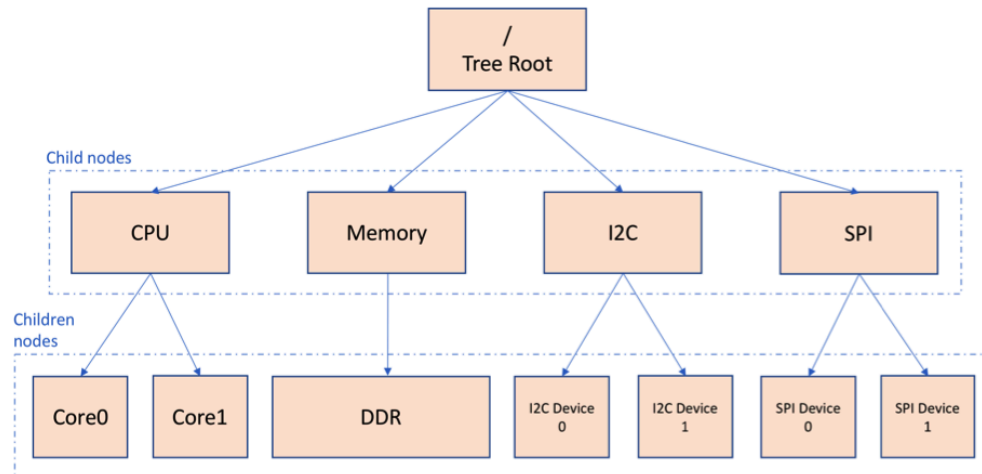


Fig. 3- Device tree em sistemas com Linux embarcado.

Fonte (imagem): [Octavosystems](https://www.octavosystems.com/)

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 **systemd**, 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 **start.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 **systemd** 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 System*, 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).

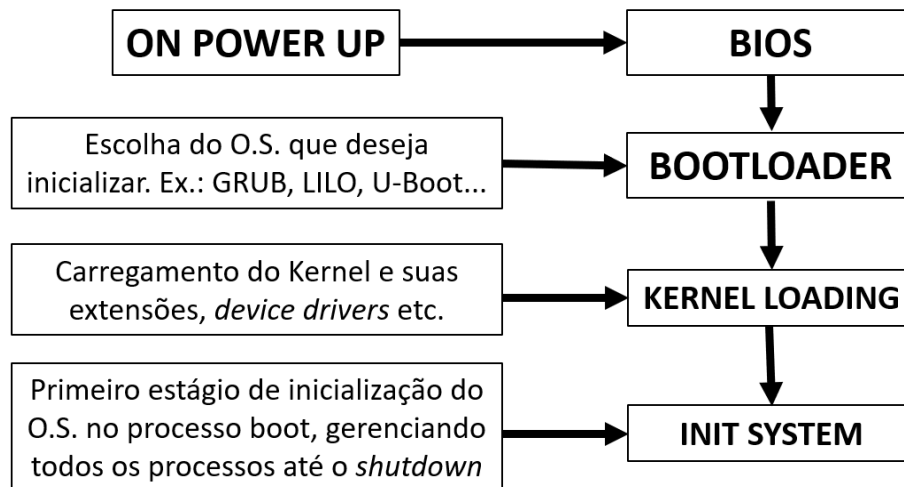


Fig. 4 - Processo “boot” e etapas de inicialização de um O.S.

Systemd

Podemos inferir que o “*systemd*” é uma ferramenta moderna que implementa o estágio “Init System” em uma “distro”(distribuição) Linux (existem outras opções: *systemv*, *Upstart*, *OpenRC*). O *systemd* 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 “*Splash Screen*” em Configurações (*sudo raspi-config > system > splash screen*)

```

[ OK ] Started System Logging Service.
[ OK ] Started Save/Restore Sound Card State.
[ OK ] Started Disable WiFi if country not set.
[ OK ] Started Avahi mDNS/DNS-SD Stack.
[ OK ] Started Login Service.
      Starting Load/Save RF Kill Switch Status...
[ OK ] Started LSB: Switch to ondemand cpu ...rnr (unless shift key is pressed).
[ OK ] Started Load/Save RF Kill Switch Status.
[ OK ] Started LSB: Autogenerate and use a swap file.
[ OK ] Started Configure Bluetooth Modems connected by UART.
[ OK ] Created slice system-bthelper.slice.
      Starting Bluetooth service...
[ OK ] Started Bluetooth service.
[ OK ] Started Raspberry Pi bluetooth helper.
[ OK ] Reached target Bluetooth.

```

Fig. 5 - Inicialização de serviços com *systemd* 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 *systemd*, 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 (< *sudo systemctl status processoX* >; < *sudo systemctl start/stop/enable processoX* >).

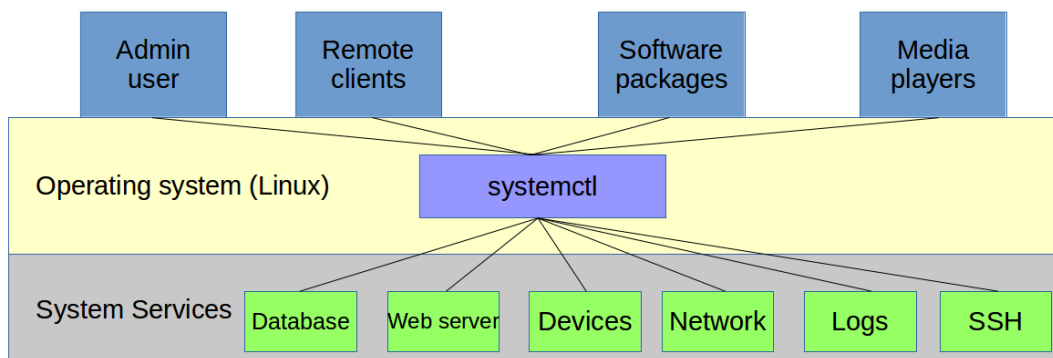


Fig. 6 - Exemplo de gerenciamento de serviços com *systemd* por meio do comando “systemctl”.

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 “**cggroups**” (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 “**cggroup** pai”, evitando confusões sobre a herança dos processos (comparativos: [systemd](#) vs. [systemv](#)).

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

<i>systemv</i>	<i>systemd</i>
Mais antigo e popular <i>init system</i> , seguindo o padrão Unix	Mais recente e moderno <i>init system</i> , usado em várias distros da atualidade (incluindo Raspbian)
Inicialização por meio de arquivos de script <i>shell</i>	Inicialização via arquivos na extensão “ <i>.service</i> ”
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.

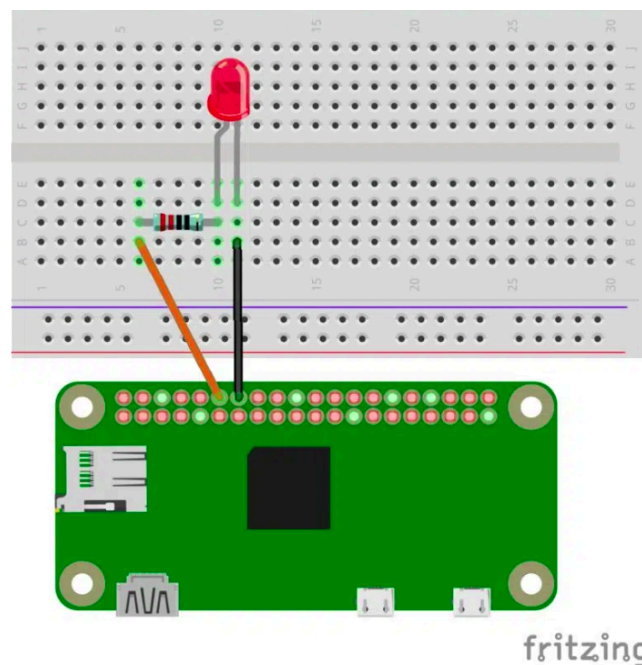


Fig. 7. Montagem prática do circuito que será usado como serviço no *systemd*

Roteiro (Parte 1):

- 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.
- 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 “*ctrl+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 *systemd* o grupo alvo no qual o serviço que deverá ser inicializado faz parte (exemplo: grupo multi-user).
- Sabemos agora, portanto, que o *systemd* 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 *systemd*, via *unit file* “*blink.service*”, para inicializar o *blink.sh*.

- Copie o arquivo “*blink.service*” para o diretório do padrão de serviços do *systemd*, 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 *systemd* 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 *systemd* 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 *systemd*. 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 “*Splash 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 status***. 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.

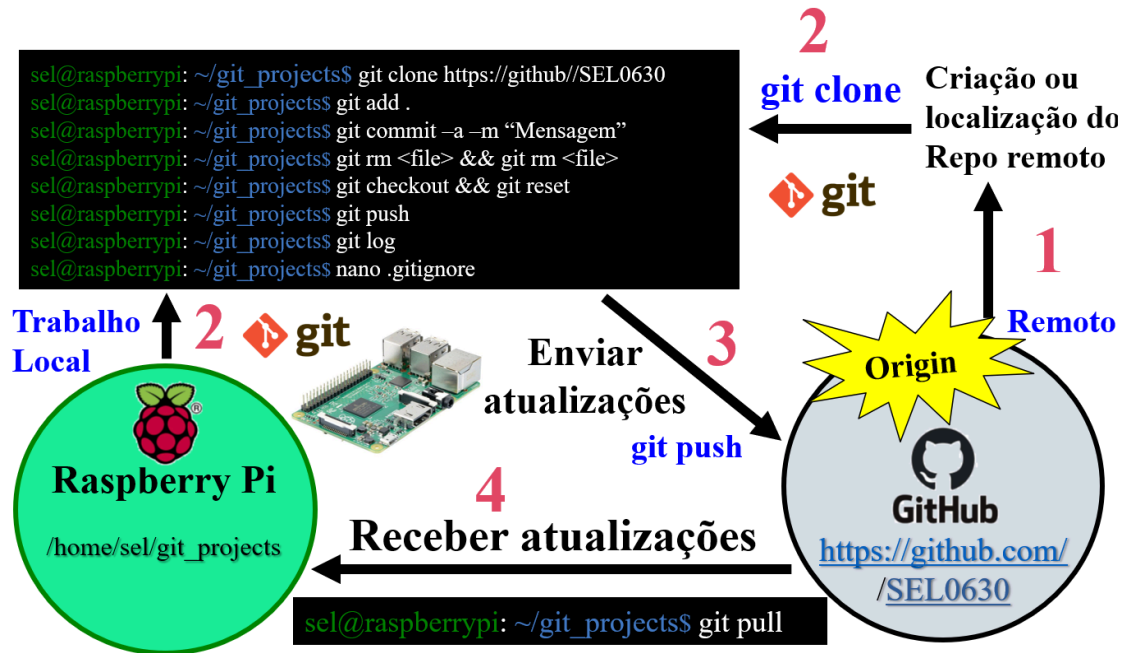


Fig. 8- Ideia do projeto com documentação usando Git/GitHub

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.