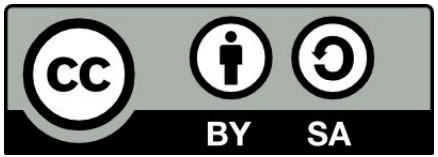
Linux Embarcado

Revisão: 25/05/2026

B2Open Systems



Informações Gerais

Cleiton Bueno

- Proprietário da **B2Open Systems**
- Usuário Linux desde 2006 e trabalhando com Linux Embarcado desde 2013
- Participamos de projetos com Linux Embarcado em IoT, Indústria 4.0, Equipamento Médico, Agro, Telecom, Radar entre outros
- Atualmente trabalhamos com Yocto Project, OpenWRT, Buildroot e desenvolvimento em C, C++ e Python

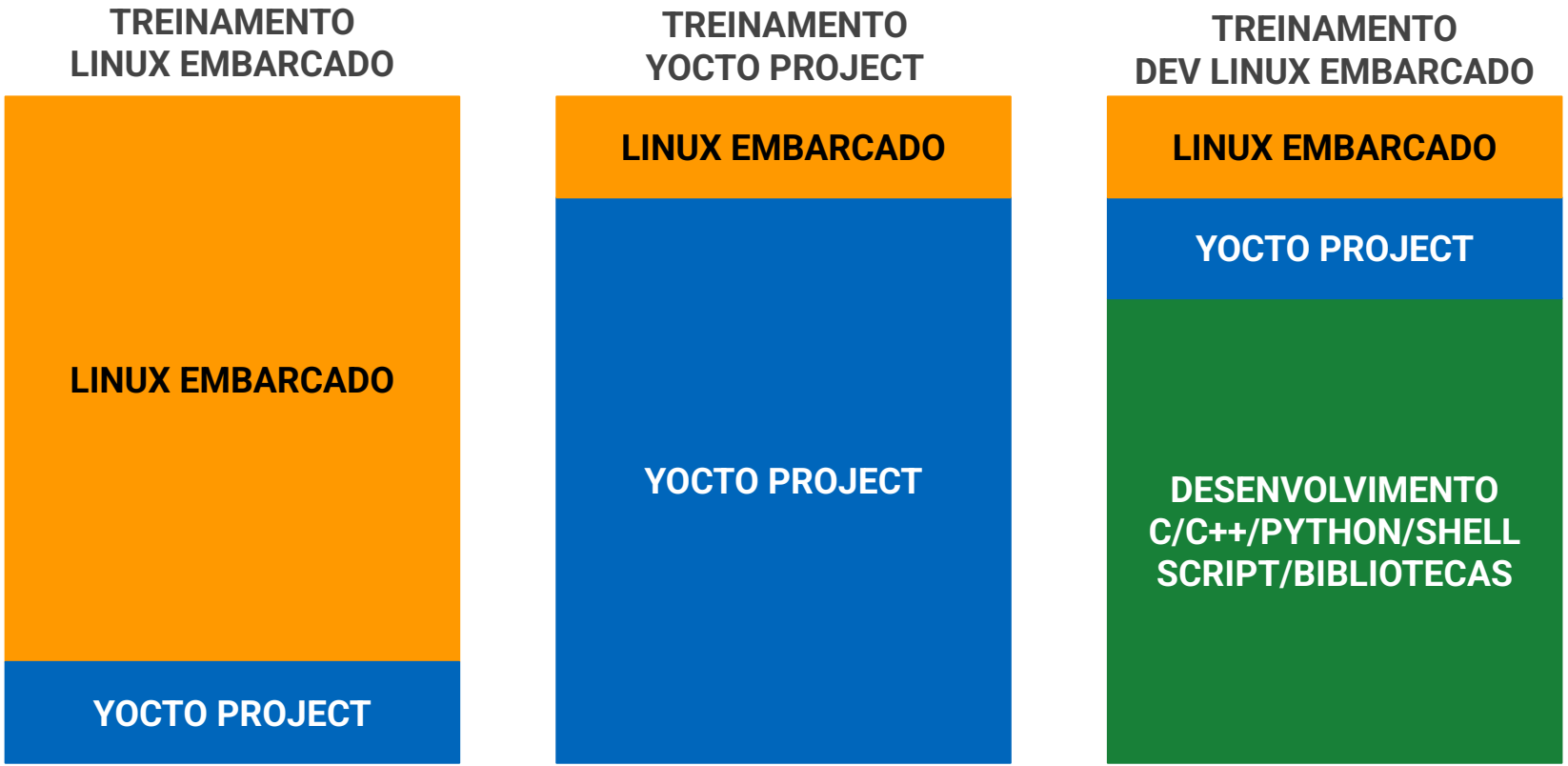
Agenda

- **Dia 1** - Introdução e estudo da anatomia de um Sistema Embarcado com Linux e seus componentes Toolchain, Bootloader e Kernel

- **Dia 2** - Linux FHS, RootFS com Busybox, Gerenciador de Serviços e Ferramenta de construção de Sistemas Linux

- **Dia 3** - Aplicações Demo, Yocto Project e Ferramentas

Treinamentos



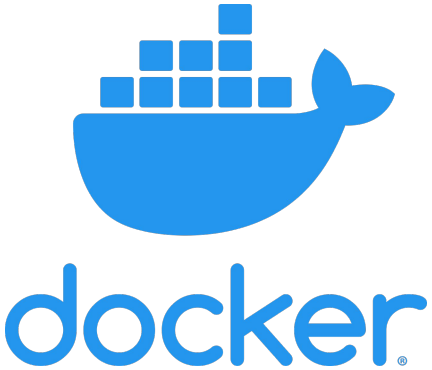
Material do Treinamento

SlidesLE.pdf - Slides do treinamento

Dockerfile - Arquivo docker para preparar o Container do treinamento

Laboratório

- Container Docker
- Todas práticas e exercícios serão dentro do Container disponibilizado via Dockerfile para preparação do Treinamento
- Uma das principais vantagens possui o mesmo ambiente/versões/configurações em diferentes Distribuições Linux
- Fácil de compartilhar com demais colaboradores

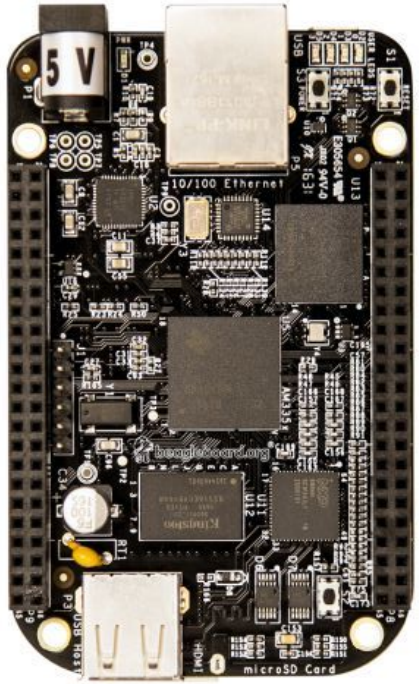


Participe!

- Não hesite em perguntar
- Sua dúvida pode ser uma questão generalizada
- Ajuda o instrutor a formatar melhor a explicação ou das mais ênfase no tópico
- Durante os exercícios da prática qualquer erro ao reproduzir comunique o Instrutor

Hardware Utilizado - BeagleBone Black

- Processador TI Sitara XAM3359AZCZ100, 1GHZ
- 2x PRU 32-bit microcontrollers
- Memória RAM 512MB DDR3L
- 4GB 8-bit eMMC OnBoard
- Controlador Ethernet 10/100
- 46 pinos GPIO
- HDMI
- 4x USB 2.0
- USB Host
- Micro SD



Mais informações: [Beaglebone Black]

Hardware Utilizado - Toradex Colibri iMX6

- NXP/Freescale i.MX6S – Solo Core, 256MB RAM e 4GB eMMC
- NXP/Freescale i.MX6DL – Dual Core, 512MB RAM e 4GB eMMC
- ARM Cortex-A9 (800MHz ~ 1GHz)
- 5x UART's, 4x SPI, 3x I2C, 2x CAN, 4x PWM
- >150 GPIO's
- GPU Vivante GC880
- Video Decode (MJPEG, MPEG-4, H.264, H.263, DivX, VC1, MPEG-2)
- Video Encode (MJPEG, MPEG-4, H.264, H.263)
- Micro SD

Mais informações: [Toradex Colibri iMX6]

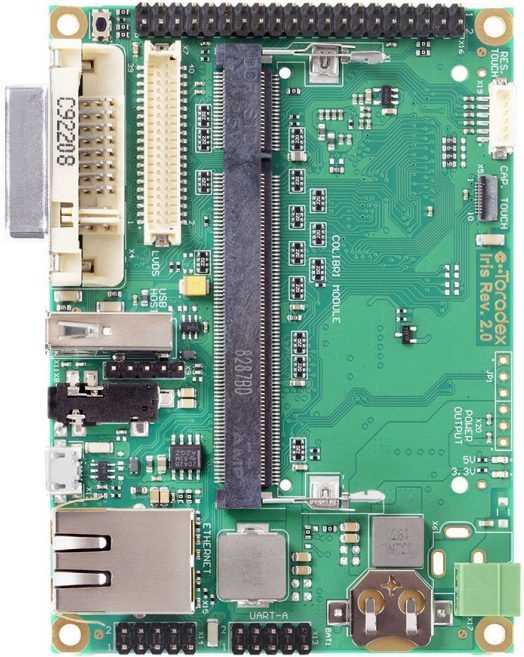
Hardware Utilizado - Toradex Colibri iMX8X

- NXP/Freescale i.MX8QX – QuadCore, 2GB RAM e 8GB eMMC
- NXP/Freescale i.MX8DX – DualCore, 1GB RAM e 4GB eMMC
- ARM Cortex-A35 (1.2GHz)
- 5x UART's, 3x SPI, 8x I2C, 3x CAN, 10x PWM
- >90 GPIO's
- GPU Vivante GC7000 Lite
- Video Decode (4K H.265 decoder*, 1080p H.264 decoder)
- Video Encode (1080p30 H.264 encoder)
- Micro SD
- Wireless Dual-Band 802.11ac / Bluetooth 5

Mais informações: [Toradex Colibri iMX8]

Hardware Utilizado - Toradex BaseBoard IRIS

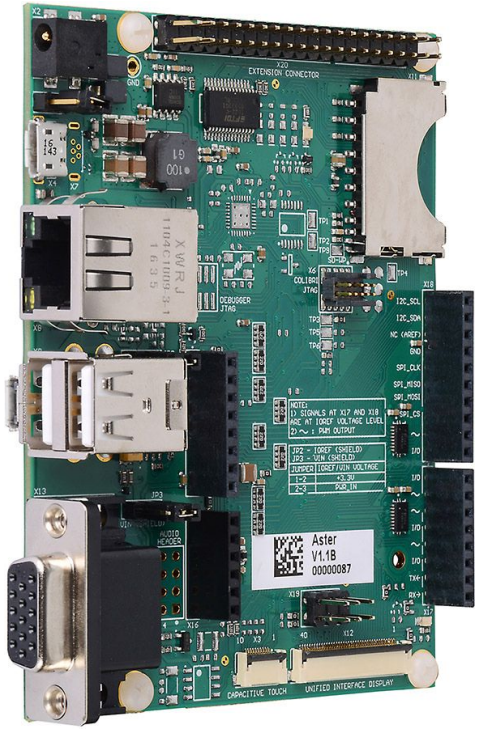
- Alimentação 6-27V DC
- 1x USB Host
- 1x USB OTG
- 3x UART's RS232
- 4x PWM
- 1x Ethernet
- 1x LVDS, 1x HDMI (Conector DVI), 1x VGA (Conector DVI)
- 1x RTC na placa
- 1x uSD
- >25 GPIO's



Mais informações: [Toradex IRIS Carrier Board]

Hardware Utilizado - Toradex BaseBoard ASTER

- Alimentação 5V DC
- 2x USB Host
- 1x USB Client
- 2x UART TTL
- 1x USB UART
- 4x PWM
- 1x Ethernet
- 1x RGB, 1x VGA (Conector DVI)
- 1x RTC na placa
- 1x SD
- >39 GPIO’s



Mais informações: [Toradex ASTER Carrier Board]

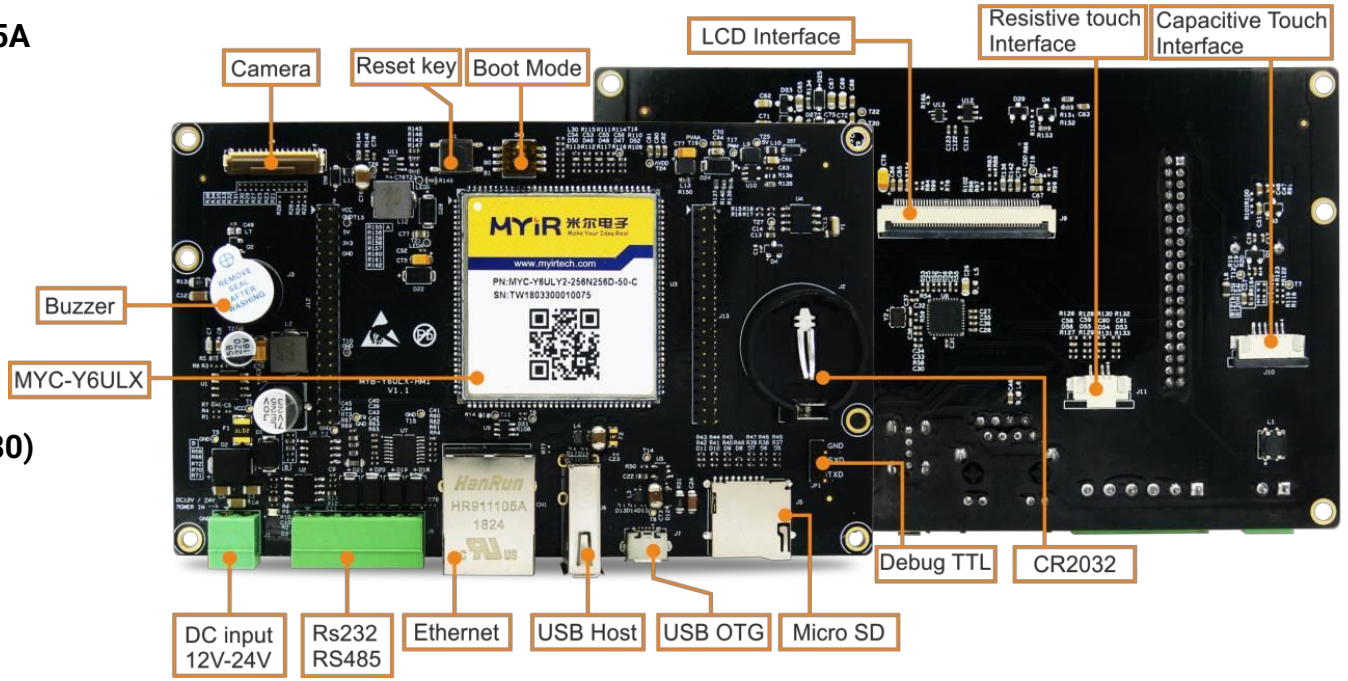
Hardware - MYIR MYC-Y6ULX

- Processador NXP iMX 6UL/6ULL ARM Cortex-A7
- Memória RAM 256MB DDR3 SDRAM
- Memória Armazenamento 256MB Nand Flash
- Conectividade de rede: FAst Ethernet 10/100
- 8x UART's
- 2x CAN
- 1x LCD 24-bit Parallel
- 2x ADC
- 2x SDIO
- USB 2.0 HS/FS / USB OTG

Mais informações: [MYC-Y6ULX CPU Module]

Hardware - MYD-Y6ULX-CHMI

- Alimentação 12V-24V DC/1.5A
- 1x USB Host
- 1x USB OTG
- 1x RS232 / 1x RS485
- 1x Serial Debug
- 1x Ethernet
- Display 7” TFT-LCD (800 x 480)
- TouchScreen Capacitivo
- 1x RTC na placa
- 1x MicroSD
- 1x Buzzer
- >97 GPIO’s



Mais informações: [MYD-Y6ULX-CHMI Display Panel]

Hardware - MYIR MYC-YF13X

- Processador ST STM32MP135 ARM Cortex-A7
- Memória RAM 256MB/512M DDR3L SDRAM
- Memória Armazenamento 256MB Nand Flash / 4G ou 8G eMMC
- Conectividade de rede: 2x Fast Ethernet 10/100
- 8x UART's
- 2x CAN
- 1x LCD 24-bit Parallel
- 2x ADC
- 1x EEPROM
- USB 2.0 HS/FS / USB OTG

Mais informações: [MYC-YF13X]

Hardware - MYD-YF13X

- Alimentação 12V
- 2x USB Host / 1x USB OTG
- 1x CAN
- 1x RS232 / 1x RS485
- 1x Serial Debug
- 2x Ethernet
- Display 7” TFT-LCD (800 x 480)
- TouchScreen Capacitivo
- 1x RTC na placa
- 1x MicroSD
- 2x Buttons (RESET e USER)
- >100 GPIO’s

Mais informações: [MYD-YF13X]

Hardware Utilizado - RaspberryPI3

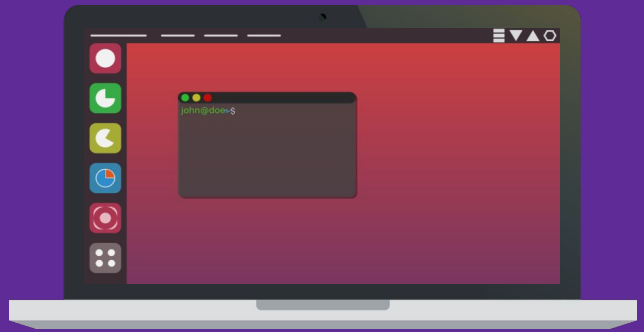
- Processador Broadcom BCM2837B0, Cortex-A53 (ARMv8) SoC de 64 bits
- Memória RAM 1GB LPDDR2 SDRAM
- Conectividade Sem Fio 2.4GHz IEEE 802.11 b/g/n/ac e Bluetooth 4.2 BLE
- Conectividade de rede: Gigabit Ethernet via USB 2.0
- Portas: GPIO 40 pinos
- HDMI
- 4x USB 2.0
- CSI (câmera Raspberry Pi)
- DSI (Display)
- Micro SD

Mais informações: [RaspberryPI Documentation]

Itens adicionais

- Conversor USB-Serial TTL 3v3
- MicroSD >8GB

Ambiente de Desenvolvimento



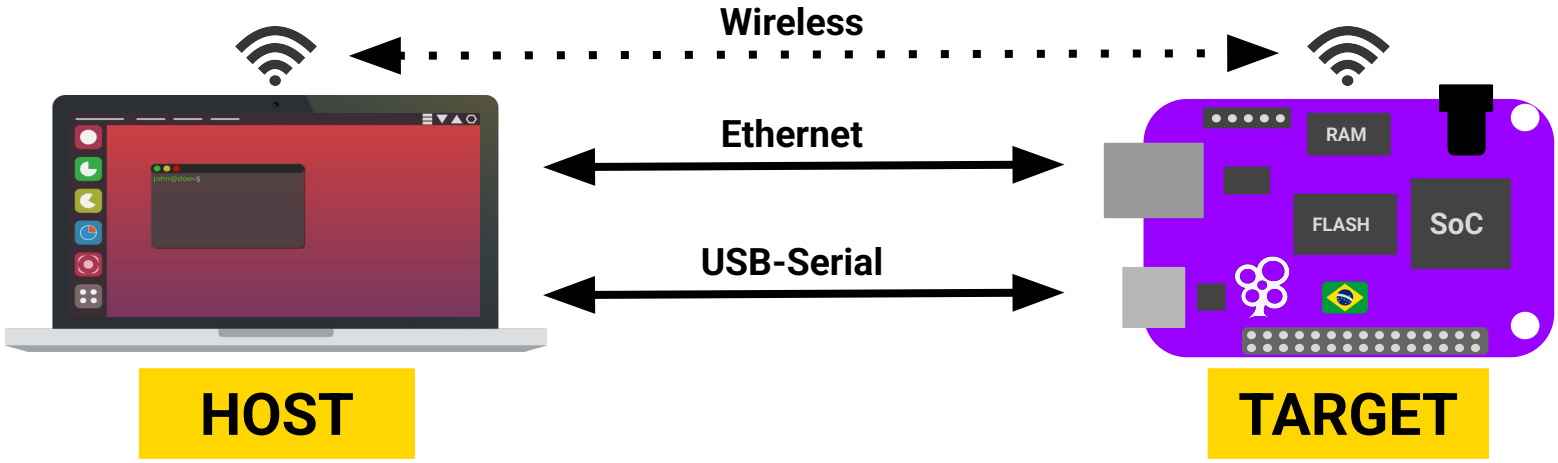
Ambiente de Desenvolvimento

- Use a Distribuição Linux de sua preferência: Debian, Ubuntu, Mint, Fedora, CentOS, Arch, ...
- Container Docker ao invés de Máquina Virtual, fácil de reproduzir, compartilhar e com menos recursos
- Todas ferramentas utilizadas são open-source
- Ambiente 100% Linux

Ambiente de Desenvolvimento

HOST Computador de Desenvolvimento

TARGET Placa Alvo, Kit de Desenvolvimento



Ambiente de Desenvolvimento

Durante o treinamento tudo é feito no terminal, algumas orientações de uso a seguir:

- Terminal como usuário comum (restrições de comandos)

```
$
```

- Terminal como super-usuário (sem restrições de comandos/permissões)

```
#
```

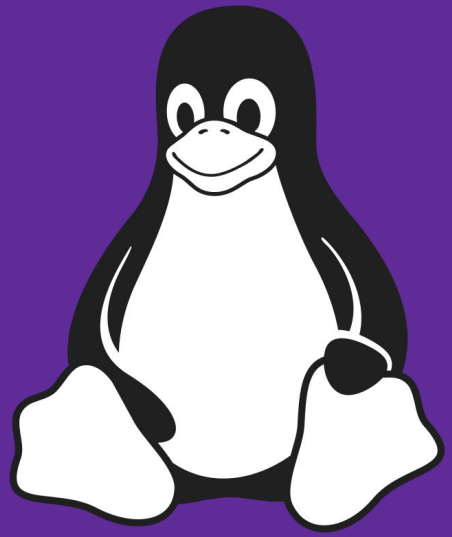
- Executando comandos root como usuário comum com **sudo**

```
$ sudo  ifconfig eth0 down
```

Laboratório

- Preparando o ambiente do Treinamento
- Prática Terminal e Comandos

Introdução ao Linux Embarcado



Do nascimento do Software Livre ao Linux

1970 – Nasce o UNIX

No Bell Labs, **Ken Thompson** e **Dennis Ritchie** desenvolvem o **UNIX**, sistema operacional multiusuário e multitarefa, escrito em linguagem C.

- Base conceitual para diversos sistemas futuros, inclusive o Linux.

1983 – Projeto GNU e Software Livre

Richard Stallman lança o projeto **GNU** (GNU's Not Unix), com o objetivo de criar um sistema UNIX-like totalmente livre.

- Ferramentas como gcc, gdb, glibc, bash são desenvolvidas.
- Tinha tudo para um Sistema Operacional, menos o Kernel. E surge o GNU HURD!
- Surge o conceito de **Licença GPL** (General Public License).

1991 – Nasce o Kernel Linux

Linus Torvalds, estudante finlandês, anuncia o desenvolvimento de um kernel livre e inspirado no MINIX.

- Em conjunto com as ferramentas GNU, nasce o sistema GNU/Linux.

Do nascimento do Software Livre ao Linux

Anos 1990 – Expansão e Consolidação

- O Linux começa a ser usado em servidores e ambientes acadêmicos.
- A comunidade cresce, e as primeiras distribuições surgem (Slackware, Debian, Red Hat).

2000 – Linux nos Sistemas Embarcados

- Empresas percebem o potencial do Linux para sistemas embarcados: flexível, sem custo de licença, e com suporte a arquiteturas diversas (ARM, MIPS, PowerPC).
- Surgem iniciativas como o BusyBox e projetos como o ELDK e uClinux.

2008 – Linux domina Dispositivos Móveis

- O Google lança o Android, baseado no kernel Linux.
- Linux passa a ser o kernel mais utilizado em smartphones, além de ganhar ainda mais espaço em dispositivos embarcados.

Do nascimento do Software Livre ao Linux

🔷 Hoje – Onipresença e Modularidade

Linux está presente em:

- Dispositivos embarcados (roteadores, automóveis, IoT)
- Smartphones (Android)
- Servidores, supercomputadores e desktops

🌐 É a base do ecossistema moderno de software livre e embarcado.

Curiosidade: O Linux era para ser chamado de "**Freax**", uma mistura de Free, Freak e Unix. Mas o Ari Lemmke do Administrador do ftp.funet.fi ao realizar o upload escolheu o nome Linux, inicialmente rejeitado por Linus por parecer egocêntrico, mas aceito com o tempo por insistência de Ari.

Confirmação no primeiro release: https://elixir.bootlin.com/linux/0.01/source/kernel/Makefile

Do nascimento do Software Livre ao Linux

UNIX (certificados)	UNIX (derivados)	UNIX-like
AIX (da IBM) HP-UX Solaris macOS (sim, oficialmente UNIX)	FreeBSD OpenBSD NetBSD macOS (Também entra aqui por baixo dos panos, pois usa base BSD + Mach)[1]	Linux Android

[1] - Apple Kernel Programming - BSP Overview

Software Livre?

- Um programa é considerado livre quando sua licença oferece a todos os seus usuários as seguintes quatro liberdades:
  - Liberdade para executar o software para qualquer propósito
  - Liberdade para estudar o software e alterá-lo
  - Liberdade para redistribuir cópias
  - Liberdade para distribuir cópias de versões modificadas

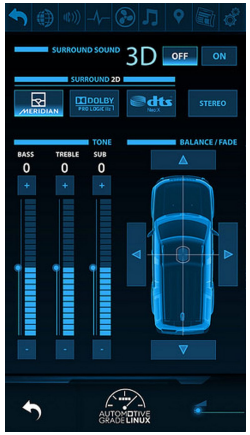
- Essas liberdades são concedidas para uso comercial e não comercial
- Implicam a disponibilidade do código-fonte, o software pode ser modificado e distribuído aos clientes

Onde Linux Embarcado está presente?


ROS


DEMO
ECG Leads Off
SpO2 High
60
30
99
80
120/90
30 10 30
120 / 100


HEXAGON
0.00ha
10.0 km/h
-0.06 m


SURROUND SOUND
3D
OFF
ON
SURROUND 2D
STEREO
TONE
BALANCE / FADE
BASS
TREBLE
SUB
0
0
0
AUTOMOTIVE GRADE LINUX


Time
Speed
Distance
Calorie
Speaker
Hand Pulse
Foldable
AUX
DC Motor


SIEMENS
SIMATIC IOT2000

Linux Embarcado

O que é Linux Embarcado?

Falar de Linux Embarcado é o mesmo de uma Distribuição Linux utilizada em Desktop ou Servidores, é utilizado o mesmo **Kernel Linux**, combinando com diversos softwares open-source e adequações para um Sistema Embarcado, na maioria das vezes com propósito bem definido!

Vantagens de utilizar Linux e Open-Source em Sistemas Embarcados?

- Reuso de Componentes
- Baixo Custo
- Controle Total
- Qualidade
- Comunidade

Linux é o Kernel!

- Sobre o Kernel: https://www.kernel.org/
- Código aberto e livre de royalties
- Portável: suporte para 23 diferentes arquiteturas (versão 5.15)
- Escalável: o mesmo kernel roda desde Servidores até SmartWatchs
- Estável: capaz de rodar por muito tempo sem precisar de um único reboot.

Linux é o Kernel!

Linux Kernel

Distribuição Cronos

Distribuição ARGO

Reuso de Componentes

- Um das principais vantagem do Kernel Linux e softwares open-source em sistemas embarcados é a capacidade de reutilizar componentes

- A comunidade já fornece uma grande gama de componentes e recursos de suporte a hardwares, protocolos de rede, bibliotecas multimídias e etc.

- Assim que um dispositivo de hardware, ou protocolo, ou um recurso é amplamente difundido, grande chance de ter componentes de código aberto que o suportem.

- Desenvolvimento mais rápido utilizando componentes prontos

- Foco no produto/software da empresa (PoC/MVP)

Baixo Custo

- O software livre pode ser duplicado em quantos dispositivos você quiser, gratuitamente.
- Se o seu sistema embarcado usa apenas software livre, você pode reduzir o custo das licenças de software a zero. Até mesmo as ferramentas de desenvolvimento são gratuitas.
- Claro, usar o Linux não é gratuito. Você ainda precisa de tempo e esforço para estudar e aprender.
- Permite um investimento maior em hardware ou treinamento para o time especializando e melhorando as habilidades em Linux.

Controle Total

- Com o código aberto, você tem o código-fonte para todos os componentes do seu sistema.
- Permite modificações, alterações, ajustes, depuração e otimização.
- Sem bloqueio de projeto por dependência de um fornecedor terceirizado ou outras prioridades.
- Permite ter controle total sobre a parte do software do seu sistema, está tudo em suas mãos!

Qualidade

- Muitos componentes de código aberto são amplamente usados em milhões de sistemas.
- Normalmente qualidade superior à que um desenvolvimento interno pode produzir, ou mesmo fornecedores proprietários.
- Claro, nem todos os componentes de código aberto são de boa qualidade, tome cuidado!
- Permite desenvolver seu sistema com componentes de alta qualidade em suas bases.

Comunidade

- Os componentes de software livre são desenvolvidos por comunidades de desenvolvedores e usuários
- Esta comunidade pode fornecer suporte de alta qualidade: você pode contatar diretamente os principais desenvolvedores do componente que está usando.
- Muitas vezes melhor do que o suporte tradicional, mas é preciso entender como a comunidade funciona para usar adequadamente as possibilidades de suporte da comunidade.
- Permite agilizar a resolução de problemas do seu projeto!

Selecionando Hardware para Sistemas Linux

É como selecionar um Microcontrolador

- Processador
- Memória RAM
- Armazenamento
- Periféricos
- Comunicação

Processador e Arquitetura

- O kernel do Linux e a maioria dos outros componentes dependentes de arquitetura suportam uma ampla gama de arquiteturas de 32 e 64 bits - (x86/x86-64, ARM, RISC-V, PowerPC, MIPS, ...)

- x86 e x86-64 - Popular em Desktops e PC's Industriais
- ARM - Popular na maioria das SBC'c e SoC (System on Chips)

Processador e Arquitetura

- Ambas as arquiteturas MMU e sem MMU são suportadas, embora as arquiteturas sem MMU tenham algumas limitações.
- Linux não suporta microcontroladores pequenos (8 ou 16 bits), talvez Zephyr seja uma alternativa.
- Além do toolchain, o bootloader e o kernel, todos os outros componentes são geralmente independentes de arquitetura

Memória RAM

- Um sistema Linux muito básico pode trabalhar com 8MB de RAM, mas um sistema mais realista de um projeto atual pode requerer mais recursos.
- Influências para mais recursos: Aplicações Gráficas, Servidor Gráfico, Aplicações Multimídia, Processamento de Imagens, IA, ML, ...
- Será visto na prática o baixo consumo de RAM!

Armazenamento

- Um sistema Linux básico pode funcionar com 4MB de armazenamento, o Kernel Linux por si só ocupa algumas centenas de KBs
- Normalmente o RootFS(Bibliotecas e Aplicações) irá demandar mais espaço de armazenamento
- Suporte **Armazenamento em Bloco**(SD/MMC/eMMC/uSD) e **Armazenamento Flash RAW**(NAND e NOR)

Comunicação

- O kernel Linux tem suporte para muitos barramentos de comunicação comuns
  - I2C
  - SPI
  - 1-wire
  - SDIO
  - PCI
  - USB
  - CAN

Comunicação

- Amplo suporte de rede
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.

Critérios para seleção

- Há diversas opções de Hardware no mercado, como:
  - **SBC** - Single Board Computer, normalmente placas populares, menor custo, tudo em uma única placa, limitado a customização
  - **EV-Boards** - Placa de desenvolvimento, completa de acesso a periféricos, display, custo maior
  - **CoM**(Computer On Module) / **SoM**(System On Module) - Hardware básico normalmente CPU, RAM e Flash, com placas customizáveis e personalizadas até por segmento ou área.

Critérios para seleção

- Certifique-se de que o Hardware que deseja utilizar tenha suporte ao Kernel e Bootloader Open-Source.

- Ter suporte nas versões oficiais dos projetos (kernel, bootloader) é muito melhor: a qualidade é melhor, novas versões estão disponíveis e versões de suporte de longo prazo estão disponíveis.

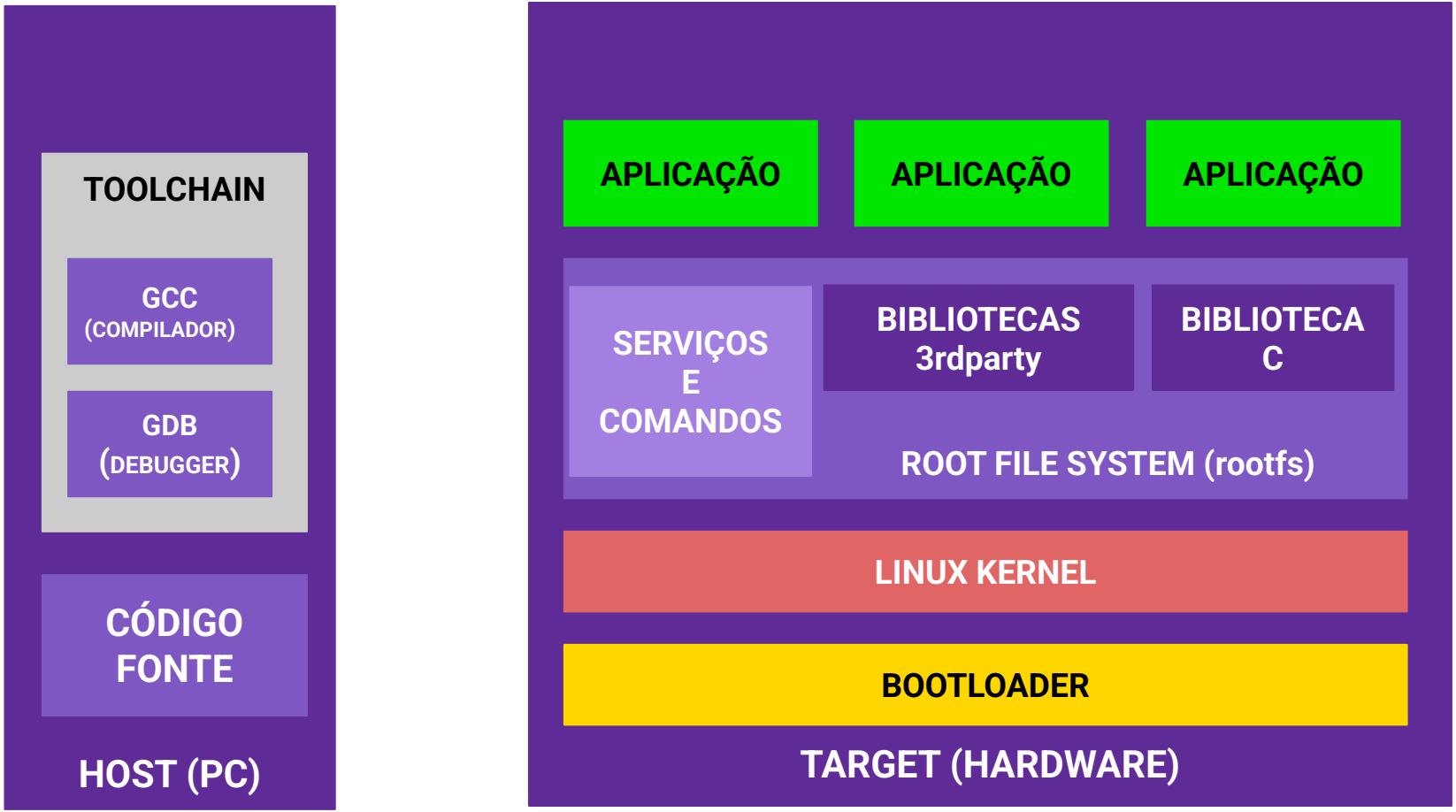
- Entre um hardware com suporte adequado no kernel oficial do Linux e hardware com suporte incompleto/desatualizado, poderá comprometer muito tempo de desenvolvimento e custo do projeto!

Arquitetura de um Sistema Linux Embarcado

Sistema Linux Embarcado, contempla...

- Toolchain
- Bootloader
- Kernel Linux
- Biblioteca C
- Bibliotecas e Aplicações

Arquitetura Sistema Linux Embarcado



Toolchain

Obtendo as ferramentas antes de iniciar os trabalhos de compilação cruzada

Definição de Toolchain

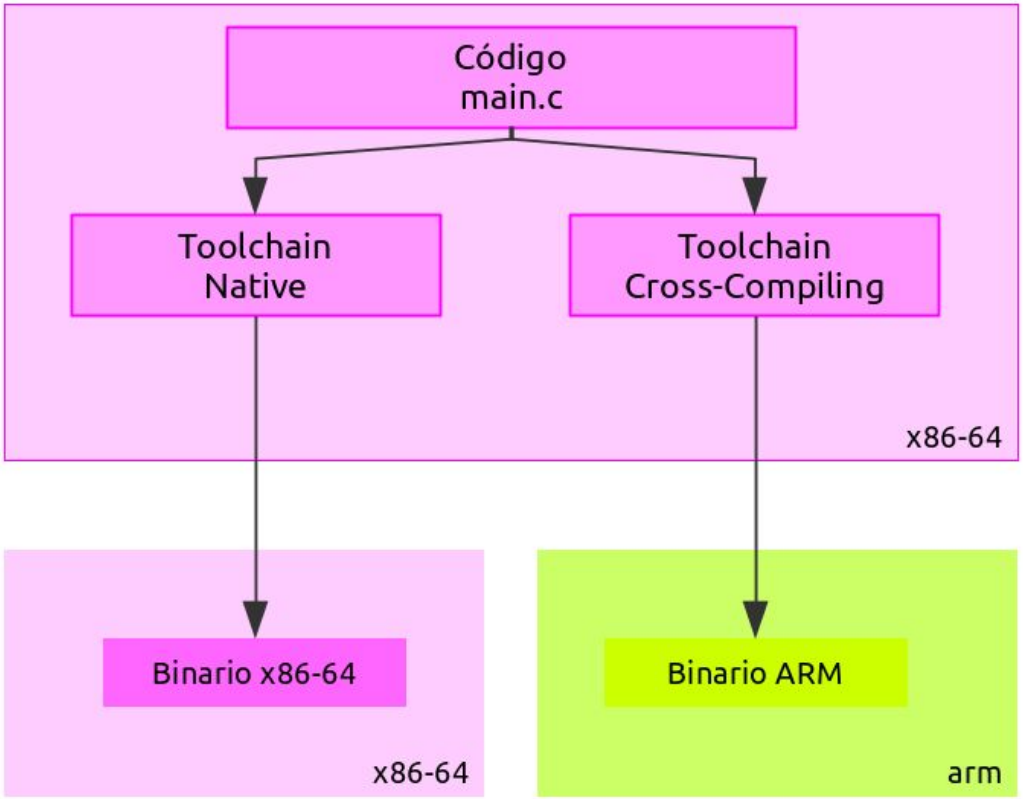
Conjunto de ferramentas que recebem um código-fonte e geram um programa final (binários) compatível com arquitetura desejada.

Tipos de Toolchain

- **Toolchain Nativo:** Utilizado em seu Desktop e gera código-nativo no próprio Host

- **Toolchain Cross-Compile:** Utilizado em Host mas para gerar código para outra plataforma(Target)

Toolchain Native x Toolchain Cross-Compiling



Máquinas na geração e uso de Toolchain

build	Onde o toolchain é construído
host	Onde o toolchain é executado
target	Onde os binários construídos pelo toolchain serão executados

Diferentes Procedimentos de Build com Toolchain

x86-64 arm mips

build host target

Native Build

Normalmente seu desktop, toolchain instalado e gera binário nativo

build host target

Cross Build

Gera toolchain para seu computador, mas gera binário para outra plataforma

build host target

Cross-Native Build

(Canadian Cross)

Gera toolchain para outra plataforma e gera binário para esta plataforma

Diferentes Procedimentos de Build com Toolchain

x86-64 arm mips

build host target

Native Build

Normalmente seu desktop, toolchain instalado e gera binário nativo

build host target

Cross Build

Gera toolchain para seu computador, mas gera binário para outra plataforma

build host target

Cross-Native Build

(Canadian Cross)

Gera toolchain para outra plataforma e gera binário para esta plataforma

Componentes de um toolchain

Binutils
(as, ld, ar, ranlib, strip, objdump, readelf, size, nm)

Kernel Headers
(Dependências da libC para “entender” as chamadas e gerar o binário corretamente)

Biblioteca C
(glibc, uclibc, musl, ...)

Compilador C/C++
(gcc e g++)

Cross-Compiling Toolchain

Depurador
(gdb)

Binutils

- Conjuntos de ferramentas para gerar e manipular binários para uma arquitetura alvo
  - **as** - O montador, gera binário baseado em um arquivo .S
  - **ld** - O linker, ligado o arquivo objeto com as bibliotecas
  - **ar, ranlib** - Para geração de bibliotecas estáticas
  - **objdump, readelf, size, nm, strings** - Para inspecionar binarios
  - **strip** - Remover partes do binário(debugging) para reduzir tamanho

GNU Binutils: https://www.gnu.org/software/binutils/

Cabeçalhos do Kernel

A **biblioteca padrão C** e os programas compilados precisam interagir com o **kernel Linux**.

➔ Para isso, precisam conhecer:

- As **syscalls disponíveis** e seus **números**.
- **Definições de constantes**.
- **Estruturas de dados**, entre outros.

Por isso, compilar a biblioteca C requer os **headers (cabeçalhos) do kernel**, e muitos aplicativos também dependem deles.

Cabeçalhos do Kernel

Os headers estão disponíveis em:

- ➔ <linux/...>
- ➔ <asm/...>
- ➔ Outros diretórios que correspondem ao conteúdo de include/uapi/ e arch/<arquitetura>/include/uapi/ no código-fonte do kernel.

Esses headers podem ser **extraídos** diretamente do código-fonte do kernel usando o comando:

```
$ make headers_install ARCH=arm INSTALL_HDR_PATH=/usr
```

Cabeçalhos do Kernel

Números de syscalls — <asm/unistd.h>

Cada arquitetura define seus próprios números:

Syscall	x86 (32 bits)	ARM (32 bits)	ARM64 (aarch64)
__NR_exit	1	1	93
__NR_fork	2	2	***(não existe)**
__NR_read	3	3	63
__NR_write	4	4	64

** **ARM64** não possui `fork` diretamente, pois utiliza apenas `clone()` e `clone3()` para criação de processos e threads.

Cabeçalhos do Kernel

Definições de constantes — <asm-generic/fcntl.h>

Usado por chamadas como **open()**:

```
#define O_RDONLY        00000000  // Somente leitura
#define O_WRONLY        00000001  // Somente escrita
#define O_RDWR          00000002  // Leitura e escrita
#define O_CREAT         00000100  // Cria arquivo se não existir
#define O_TRUNC         00001000  // Trunca o arquivo ao abrir
#define O_APPEND        00002000  // Adiciona ao final do arquivo
#define O_NONBLOCK      00004000  // I/O não bloqueante
#define O_SYNC          00010000  // Escrita síncrona
```

Essas definições passam por:

<linux/fcntl.h> → <asm/fcntl.h> → <asm-generic/fcntl.h>

Cabeçalhos do Kernel

Estruturas de dados — **<asm/stat.h>**, **<linux/utsname.h>**, etc

Exemplo de estrutura utsname usada para obter informações do Kernel, usada pela syscall **uname()**:

```
struct utsname {
    char sysname[ ];        // Nome do sistema operacional
    char nodename[ ];       // Nome da máquina
    char release[ ];        // Versão do kernel
    char version[ ];        // Compilação do kernel
    char machine[ ];        // Arquitetura da máquina
};
```

Compilador GCC

- O **GCC** (GNU Compiler Collection) é o compilador do projeto GNU.
  https://gcc.gnu.org/

- Compatível com diversas linguagens de programação, incluindo **C, C++, Objective-C, Fortran, Ada, Go e D**.

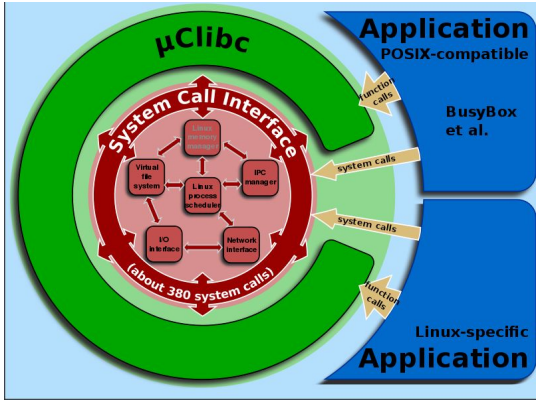
- Pode gerar código para diversas arquiteturas, incluindo **ARM, AVR, Blackfin, MIPS, PowerPC, x86, etc**.

Bibliotecas C

- O que é a biblioteca C?
  - Interface entre as aplicações e o kernel.
  - Toda implementação de C vem com biblioteca padrão de funções pré-definidas
  - Note que, em programação, uma biblioteca é uma coleção de funções
  - As funções comuns a todas as versões de C são conhecidas como Biblioteca Padrão C (open, close, exit, read, write, malloc, ...)

Bibliotecas C

- O que é a biblioteca C?
  - API para o desenvolvimento de aplicações.
  - O toolchain depende da biblioteca C, já que ele irá linká-la com sua aplicação para gerar os binários para a placa alvo.
  - Bibliotecas C populares disponíveis: glibc, uClibc-ng, musl, dietlibc, klibc, bionic, etc.



Fonte:
Wikipedia(https://bit.ly/2zrGve2)

Principais Bibliotecas C disponíveis

glibc eglibc	LGPL	**Biblioteca C do projeto GNU** Foco em performance e produtividade, principal biblioteca em Desktop e Servidores Linux
uclibc-ng	LGPL	**Mais leve e foco para sistemas embarcados** Fork do uClibc, build menor que glibc, alguns recursos podem não estar disponíveis como funções de tempo-real
musl	MIT	**Biblioteca leve, rápida e simples** Projeto escrito com foco em Sistemas Embarcados, ótima opção para gerar binários linkado estaticamente
klibc	BSD/GPL v2	**Biblioteca extremamente pequena e limitada** Projeto escrito do zero, foco em tamanho e limitado para uso básico como initramfs

Escolhendo o Toolchain

- Toolchain Pronto
- Gerando o próprio Toolchain

Toolchain Pronto

- Fácil instalação - comum instalação via gerenciador de pacotes ou download direto, configurar variável **PATH**
- Principal vantagem instalar e já sair utilizando
- Desvantagem é não permitir customização ou otimizar
- Algumas opções:
  - Linaro
  - Code Sourcery
  - Bootlin GNU toolchain

Gerando o próprio Toolchain

- Gerar o próprio toolchain não é uma tarefa fácil, muitas dependências, ferramentas, patches e configurações
- Mas hoje há ferramentas que facilitam este trabalho
- Algumas são: **Crosstool-ng**, **Buildroot** e **Yocto Project**

Principais Bibliotecas C disponíveis

Crosstool-ng	**Suporta uClibc, glibc e musl** Sistema de configuração baseado em menuconfig
Buildroot	**Suporta uClibc, glibc e musl** Sistema baseado em Makefiles
Yocto Project	**Suporta glibc e musl** Com mais recursos, mais complexo de configuração

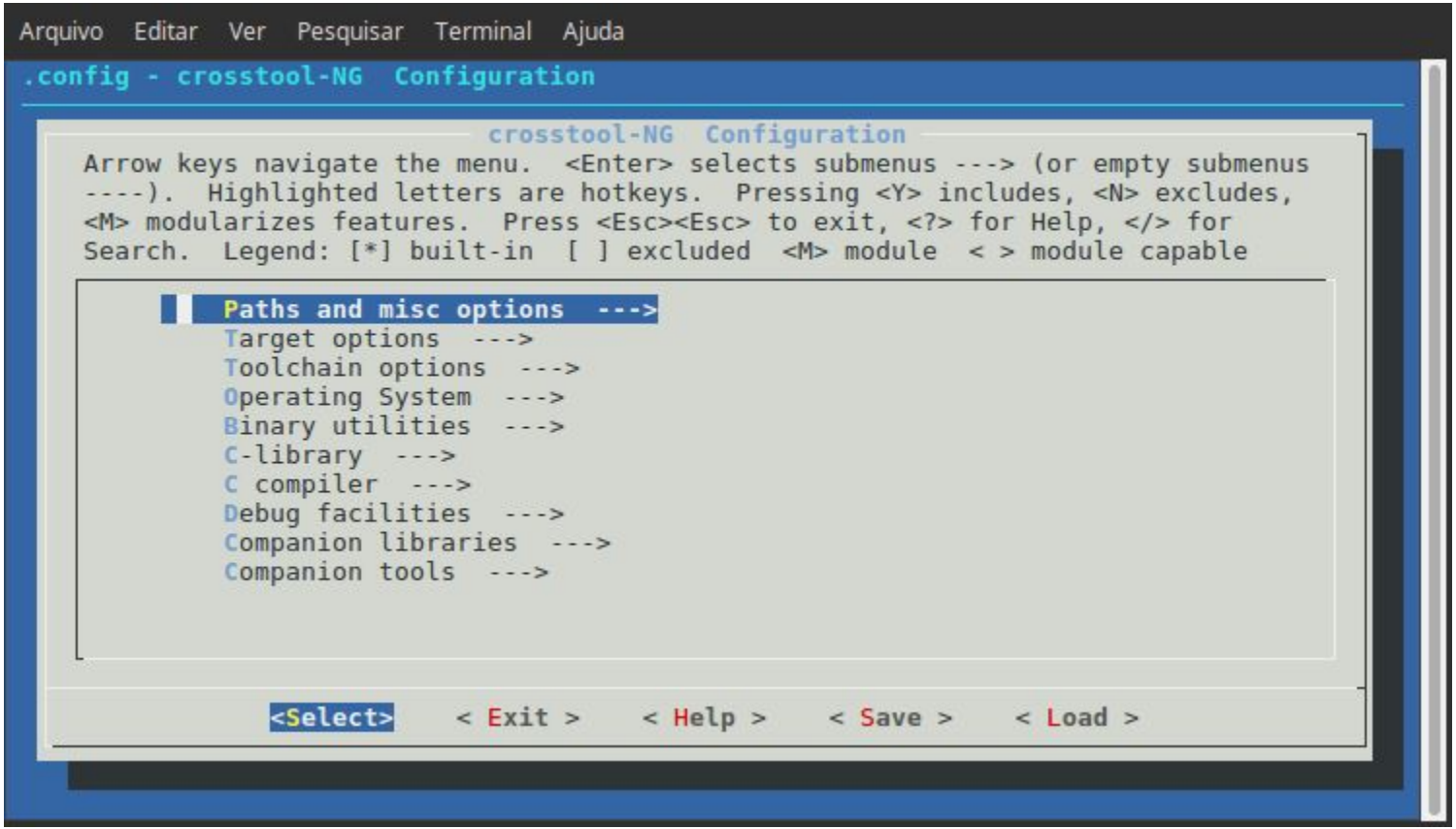
Principais Bibliotecas C disponíveis

Crosstool-ng

Suporta uClibc, glibc e musl
Sistema de configuração baseado em menuconfig

- Principal ferramentas open-source para geração de toolchains
- Possibilidade de gerar toolchains para diferentes arquiteturas: **x86, x86-64, MIPS, ARM**

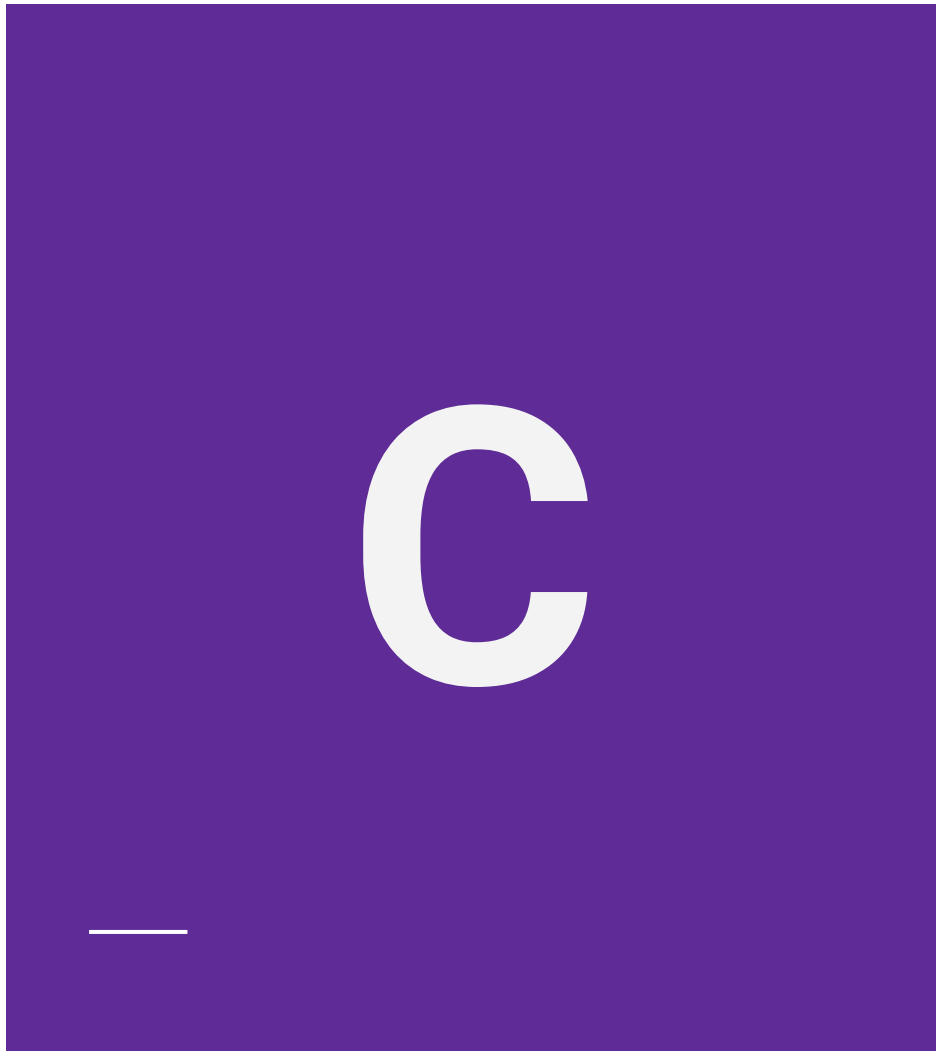
Principais Bibliotecas C disponíveis



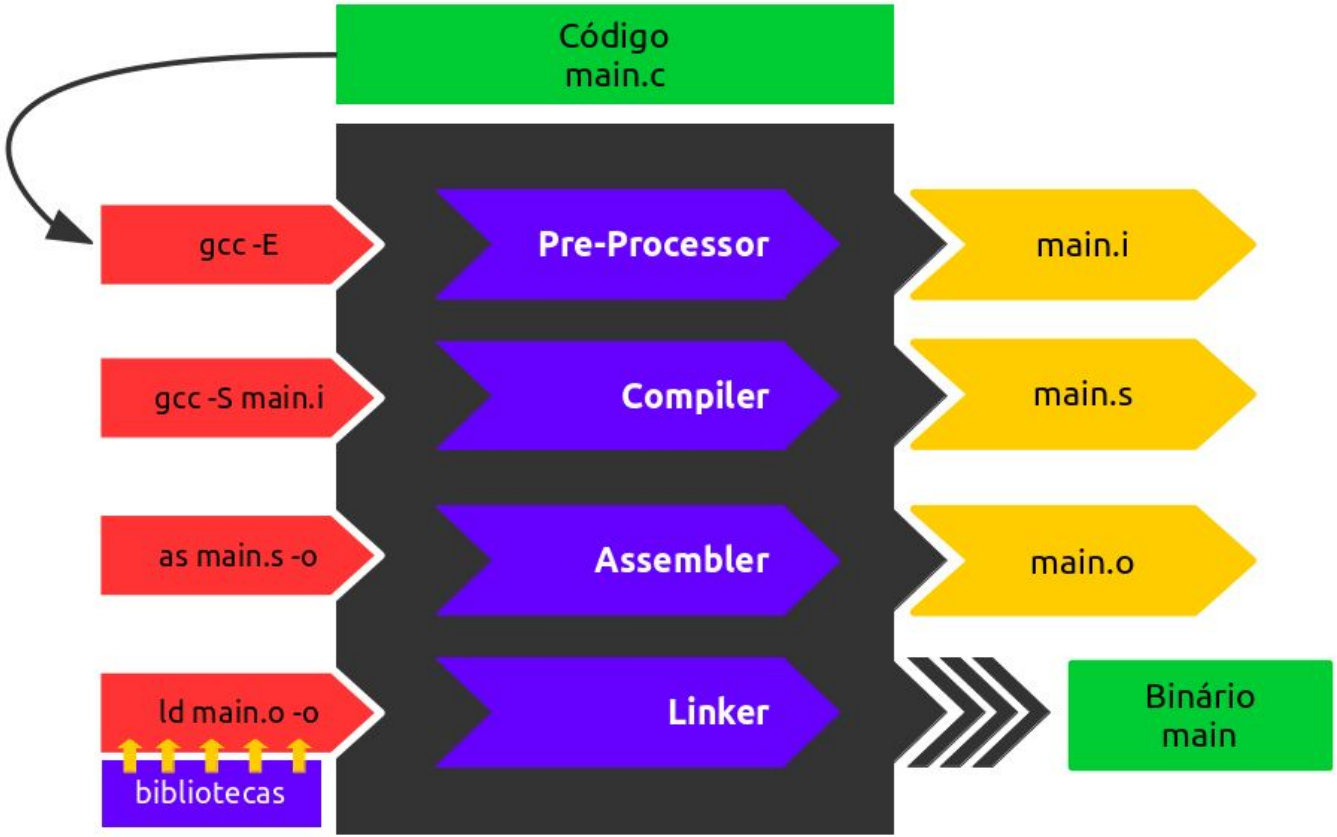
Laboratório

- Gerando o próprio toolchain com **Crosstool-NG**

Como um programa em C é compilado?



Como um programa C é compilado?



Como um programa C é compilado?

1. **Pre-processor**(Pre-Processamento): Através do pré-processador GNU C (gcc), que inclui os cabeçalhos (#include) e expande as macros (#define).

Como um programa C é compilado?

2. **Compiler**(Compilador): O compilador (gcc) compila o código-fonte pré-processado em código assembly para um processador específico. A opção -S gera código assembly, um arquivo main.s será gerado!

Como um programa C é compilado?

3. **Assembler**(Montador): O assembler (as) converte o código do assembly em código de máquina no arquivo objeto "main.o".

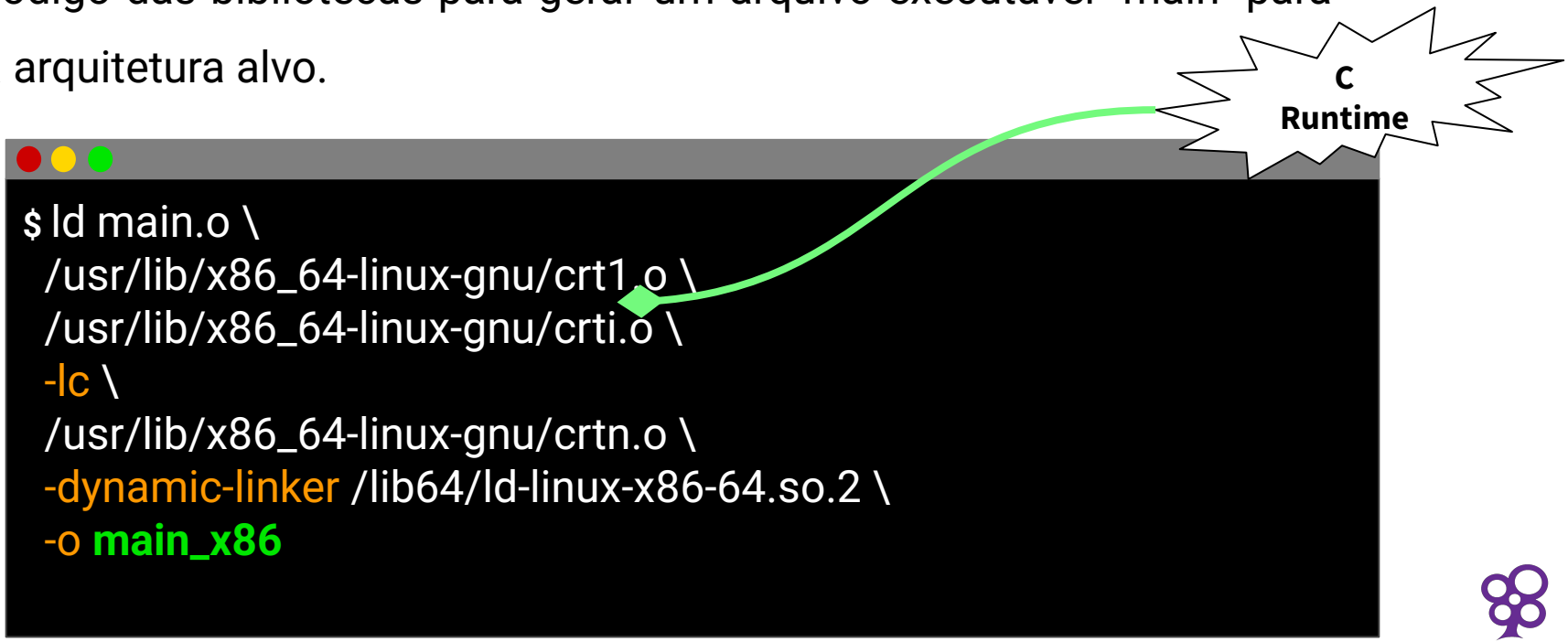
Como um programa C é compilado?

4. **Linker**(Ligador): Por último, o linker(ld) vincula o código do objeto ao código das bibliotecas para gerar um arquivo executável "main" para a arquitetura alvo.

```
$ ld main.o \
  /usr/lib/x86_64-linux-gnu/crt1.o \
  /usr/lib/x86_64-linux-gnu/crti.o \
  -lc \
  /usr/lib/x86_64-linux-gnu/crtn.o \
  -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
  -o main_x86
```

Como um programa C é compilado?

4. **Linker**(Ligador): Por último, o linker(ld) vincula o código do objeto ao código das bibliotecas para gerar um arquivo executável "main" para a arquitetura alvo.



Exemplo Código em C

Exemplo de Código

Um código para listar números de syscalls e verificar a arquitetura dinamicamente:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <sys/utsname.h>
#include <unistd.h>

int main() {
    struct utsname uts;
    if (uname(&uts) == -1) { perror("[B2Open] - Erro ao obter informacoes do sistema"); return EXIT_FAILURE; }

    // Imprimir no terminal a arquitetura em execucao e os numeros das syscalls abaixo
    printf("[B2Open] Arquitetura           : %s\n", uts.machine);
    printf("[B2Open] Numero da syscall read  : %d\n",  __NR_read);
    printf("[B2Open] Numero da syscall write : %d\n",  __NR_write);
    printf("[B2Open] Numero da syscall open  : %d\n",  __NR_open);
    printf("[B2Open] Numero da syscall close : %d\n",  __NR_close);

    return 0;
}
```

Exemplo Código em C

Compilando Código:

```
$ gcc info_kernel.c -o info_kernel
```

Executando:

```
$ ./info_kernel
[B2Open] Arquitetura          : x86_64
[B2Open] Numero da syscall read  :  0
[B2Open] Numero da syscall write :  1
[B2Open] Numero da syscall open  : 2
[B2Open] Numero da syscall close : 3
```

Laboratório

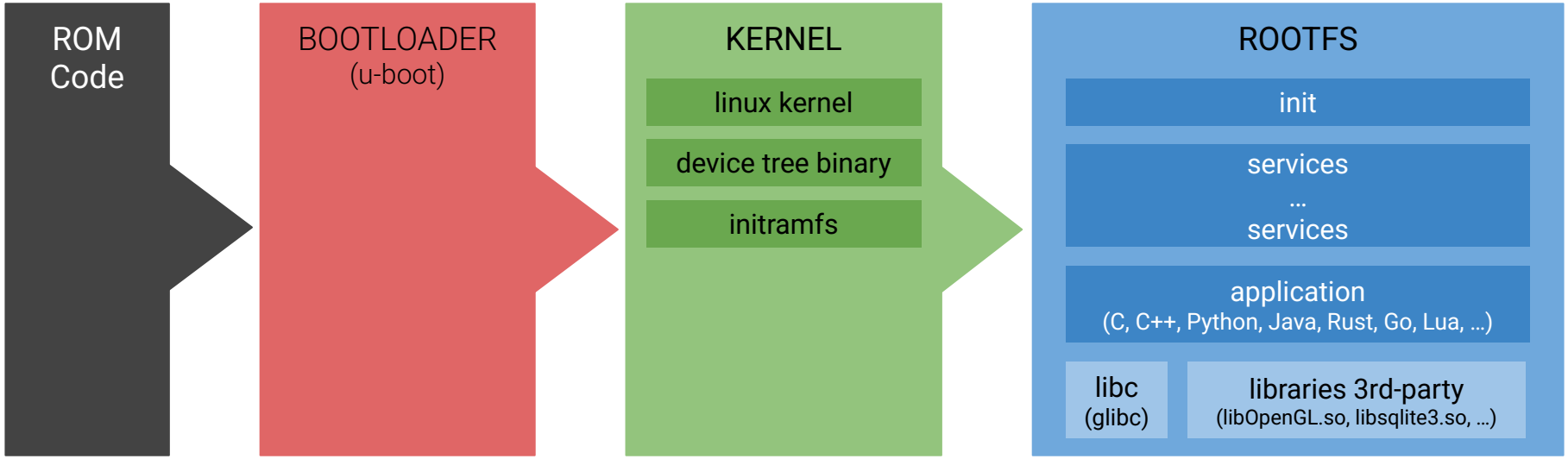
Prática 01

- Compilando um código nativo e um para ARM

Sequência de inicialização

- **Bootloader:** Inicialização básica do hardware e carregamento do Kernel Linux.
- **Linux Kernel:** Inicializa o restante do hardware e configura de acordo com device-tree, na sequência chama o init.
- **RootFS:** Aplicações, Bibliotecas, Serviços(inclusive o init) e ferramentas.

Sequência de Inicialização



Bootloader

Bootloader

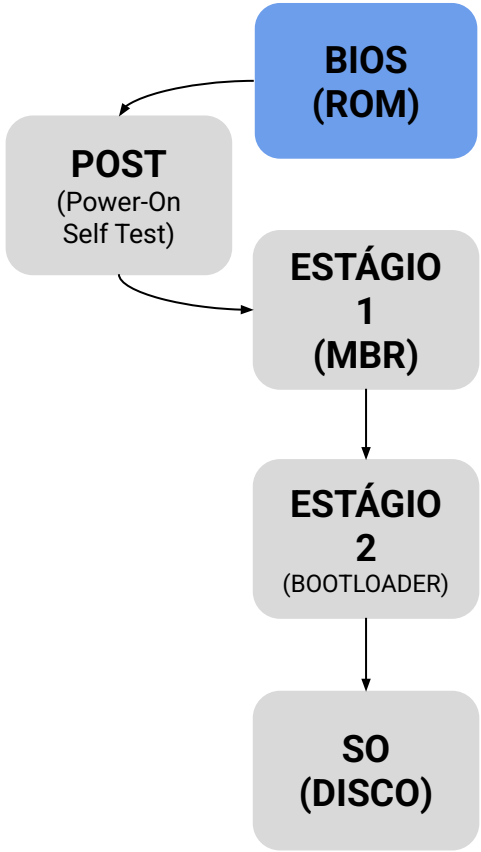
- Uma das principais funções do bootloader é:
  - Inicialização básica do Hardware (CPU, Controladora da Memória RAM, unidades de armazenamento, etc)
  - Carregar outro binário normalmente o Kernel para continuar a inicialização
  - Passar o controle da CPU para o novo binário

Bootloader

- Além dessas funções básicas, a maioria dos bootloaders fornece um shell que permite diversas operações:
  - Variáveis Ambiente, Manipulação de RAM e Flash, Inspeção de Memória, Diagnóstico de Hardware e Testes, passagem de parâmetros para o Kernel

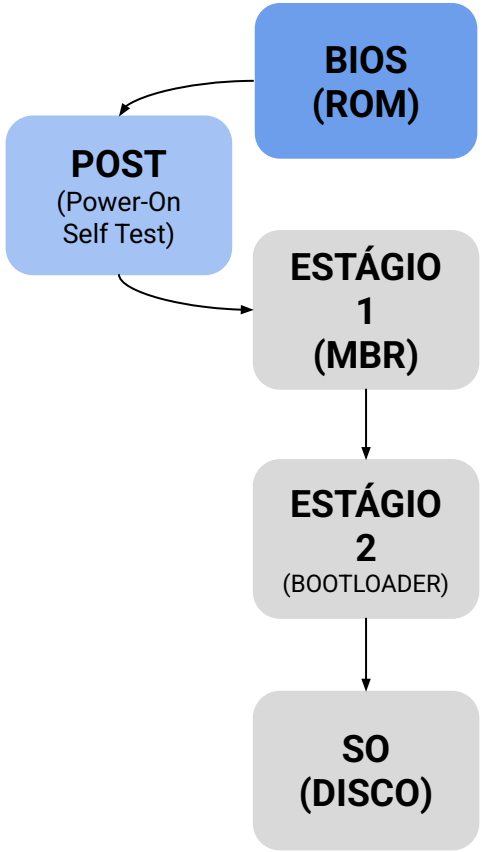
Bootloader x86

- Em plataformas x86 o bootloader reside em uma memória não-volátil, popularmente chamado de **BIOS**(Basic Input/Output System).
- Há **BIOS** está em ROM e é fornecida pelo fabricante.



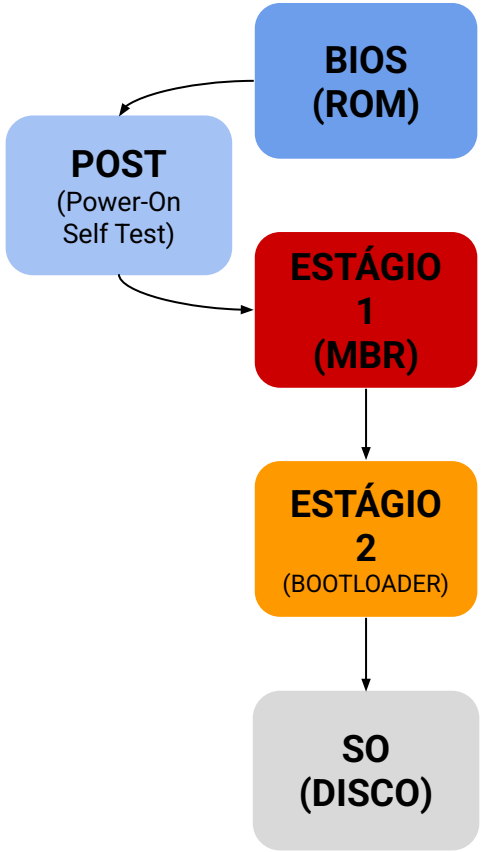
Bootloader x86

- Em plataformas x86 o bootloader reside em uma memória não-volátil, popularmente chamado de **BIOS**(Basic Input/Output System).
- Há **BIOS** está em ROM e é fornecida pelo fabricante.
- O **POST** inicializa a Memória RAM, CPU, Dispositivos de Entrada e Saída.



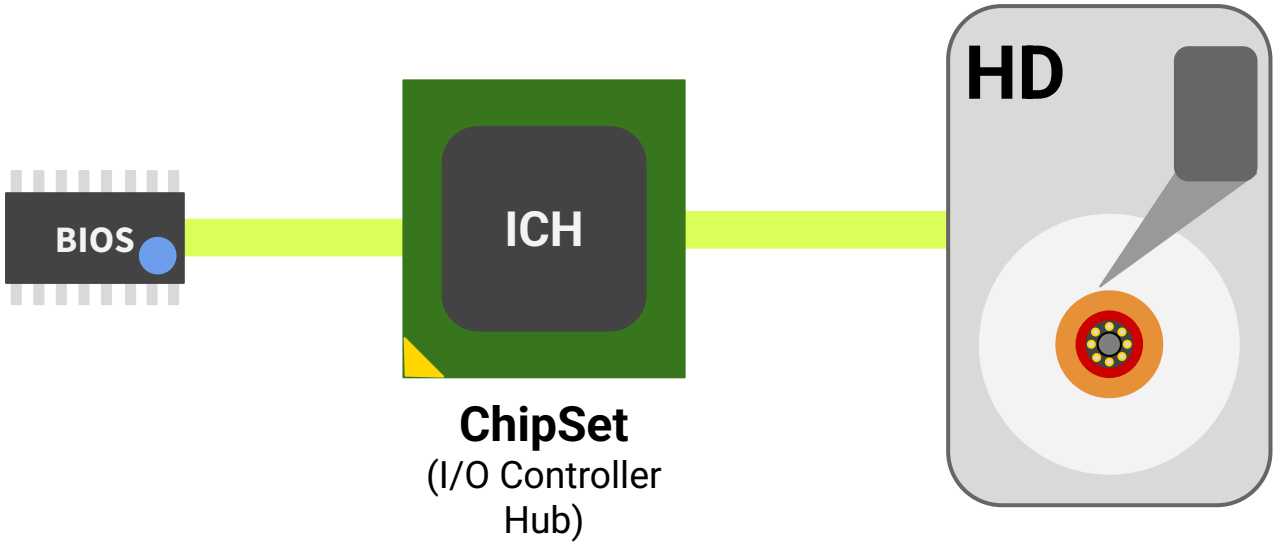
Bootloader x86

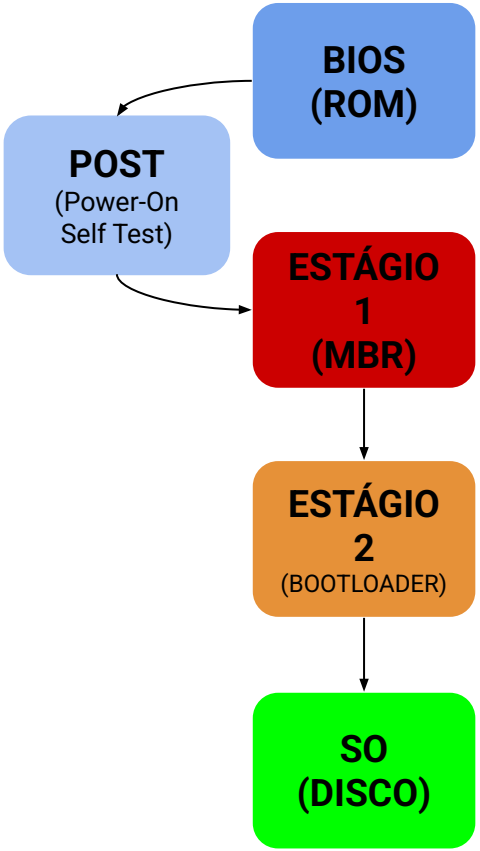
- Em plataformas x86 o bootloader reside em uma memória não-volátil, popularmente chamado de **BIOS**(Basic Input/Output System).
- Há **BIOS** está em ROM e é fornecida pelo fabricante.
- O **POST** inicializa a Memória RAM, CPU, Dispositivos de Entrada e Saída.
- Lê os primeiros 512 bytes do disco, chamado **MBR**(Master Boot Record), carregando tabela de partições e localiza o Bootloader, por exemplo **GRUB,** que seria o **Estágio 2**.



Bootloader x86

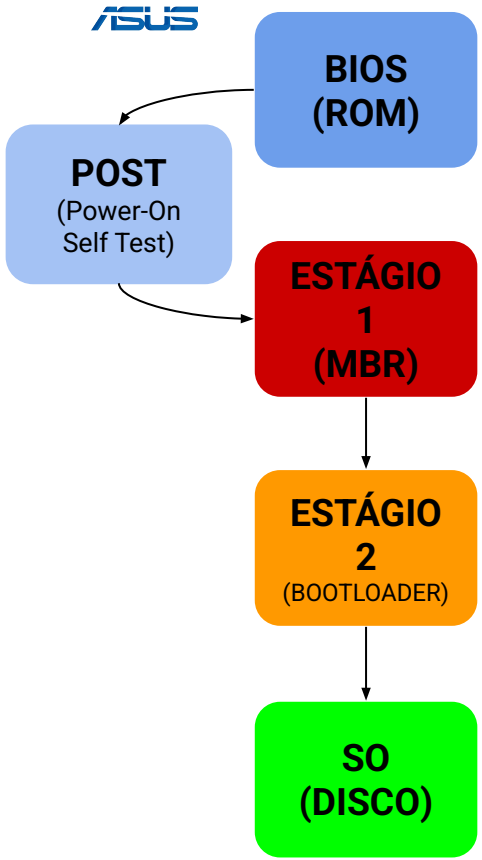
- O **Estágio 2** consegue carregar o Kernel do Disco e realizar a inicialização completa do SO.





Bootloader x86

- Principal Bootloader utilizado em x86/x86-64 - GRUB:
  - GRUB(Grand Unified Bootloader) - Site GRUB
  - Opção de Dual-Boot
  - Suporta Boot-Seguro e UEFI
  - Suporta diversos Sistemas de Arquivos como: Linux(EXT2/EXT3/EXT4), DOS(FAT12/FAT16/FAT32/exFAT), NTFS, ReiserFS, RomFS, etc.



Bootloader ARM

- A CPU possui um código de inicialização integrado na ROM.
  - ROM Code em iMX6
  - Cada CPU possui seus detalhes e particularidades do fabricante
- Este ROM Code é capaz de carregar um Bootloader de primeiro estágio em um dos dispositivos de armazenamento em uma SRAM interna (DRAM ainda não inicializada).
- O dispositivo de armazenamento pode ser normalmente: MMC, NAND, Flash SPI, etc.

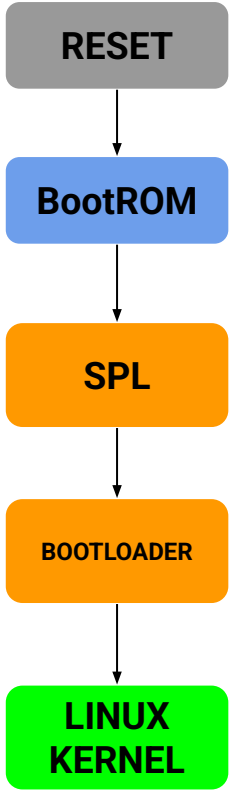
Bootloader ARM

- O bootloader de **primeiro estágio** é:
  - Tamanho limitado devido a restrições de hardware (tamanho SRAM)
  - Fornecido por U-Boot (chamado de Secondary Program Loader - SPL) ou pelo Fabricante do CPU
- Este Bootloader de **primeiro estágio** deve inicializar DRAM e outros dispositivos de hardware e carregar um Bootloader de **segundo estágio** em DRAM.

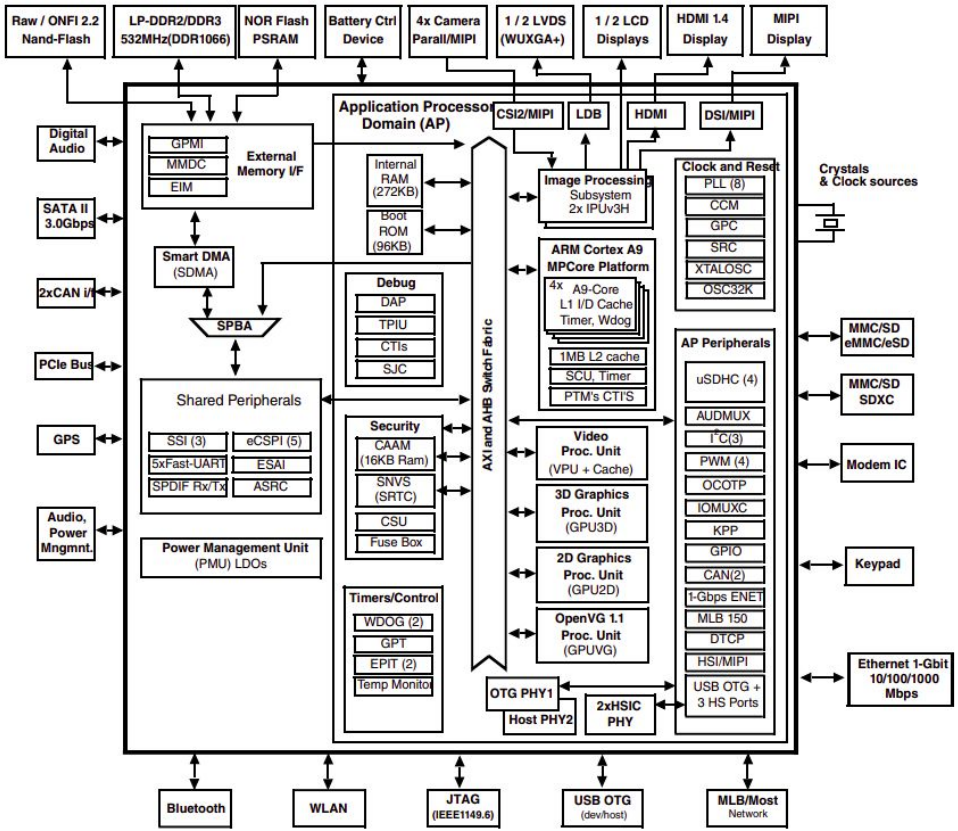
Bootloader ARM

- O Bootloader de **segundo estágio** é mais poderoso, com mais recursos e até prompt para iteração.

- É com o Bootloader de **segundo estágio** que o Kernel Linux é carregado e iniciado o boot do SO, neste momento o Bootloader não atua mais.

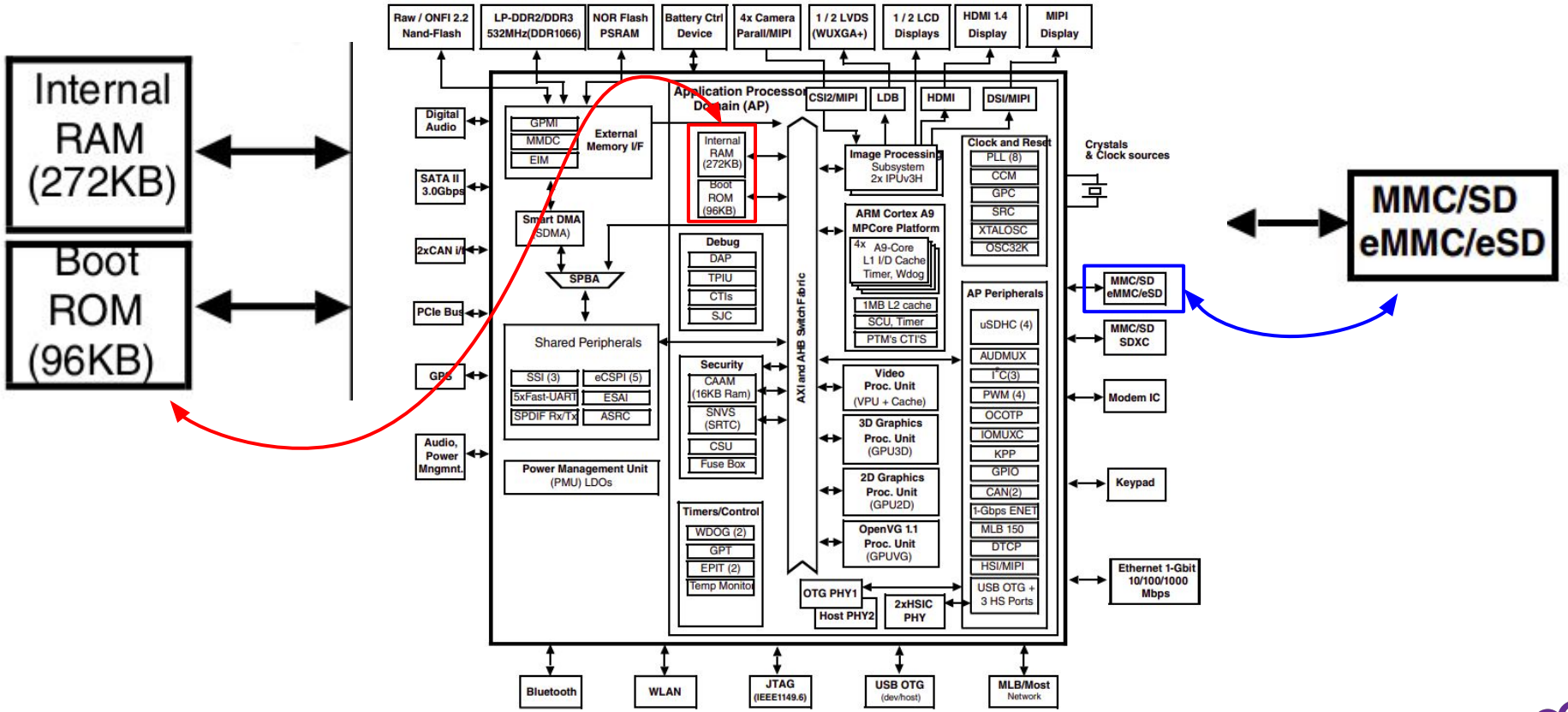


Bootloader ARM - Família iMX6



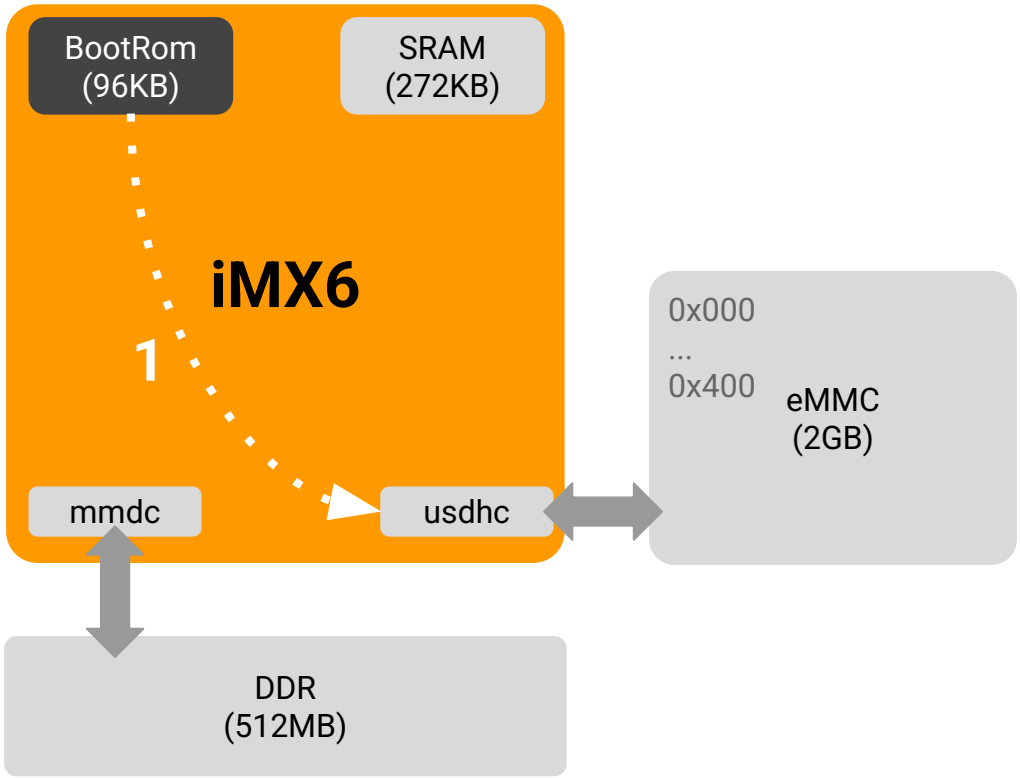
Referência: [IMX6DQRM Reference Manual]

Bootloader ARM - Família iMX6



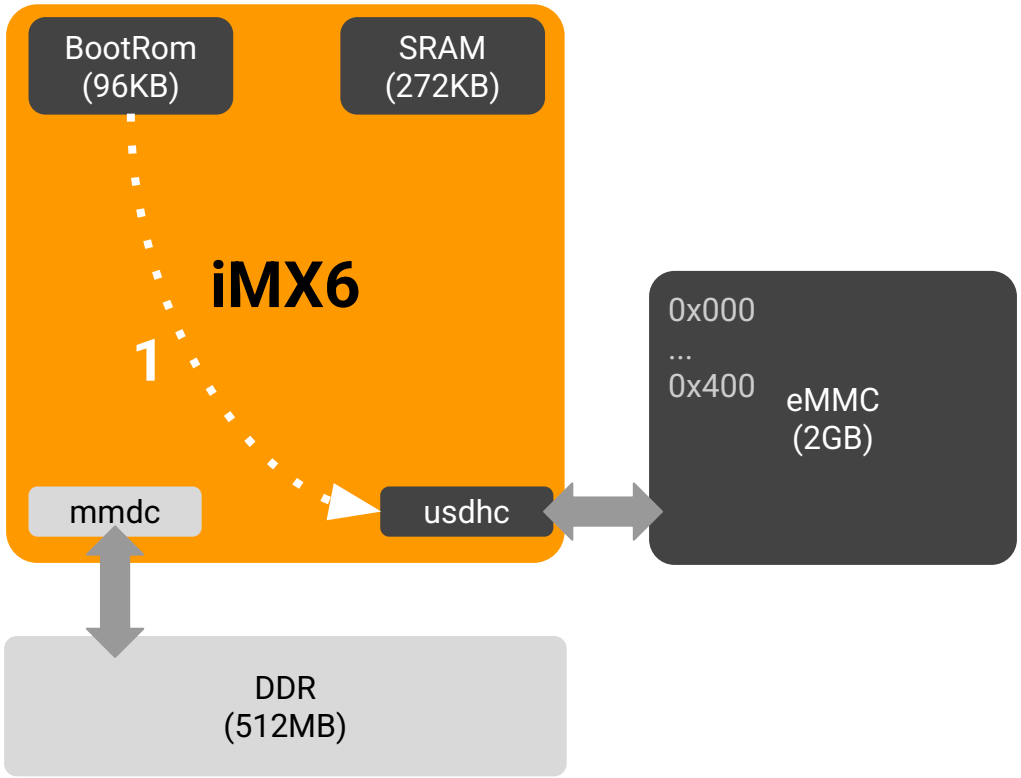
Referência: [IMX6DQRM Reference Manual]

Bootloader ARM - Família iMX6



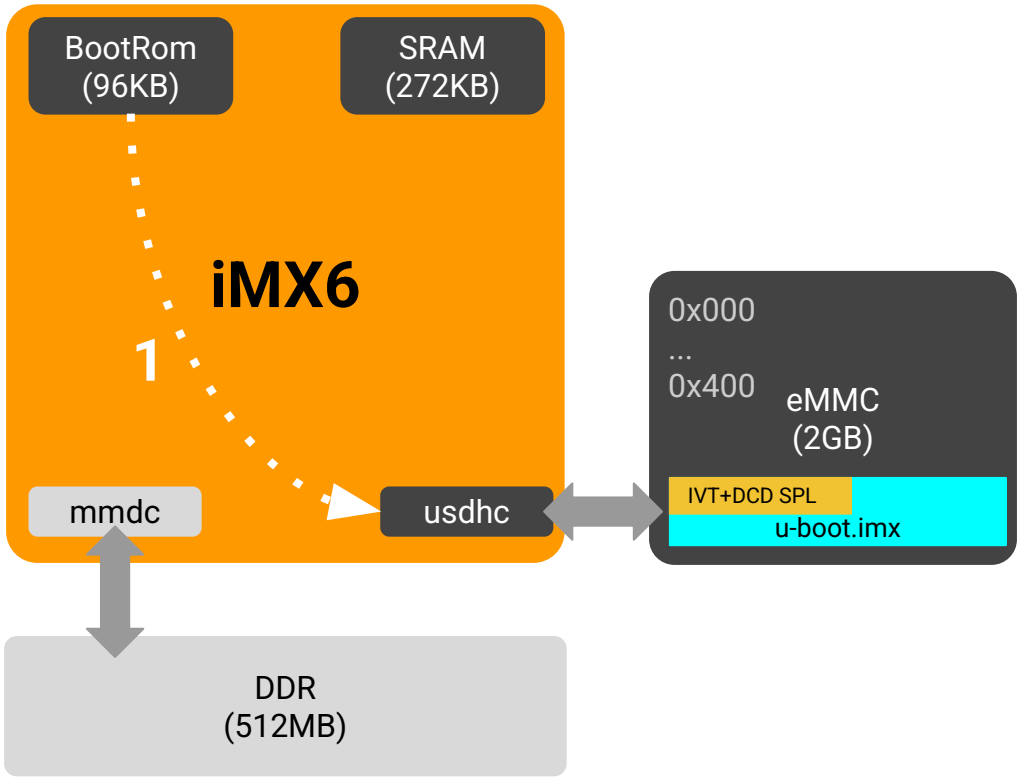
1. BootROM inicia e prepara a SRAM e USDHC.

Bootloader ARM - Família iMX6



1. BootROM inicia e prepara a SRAM e USDHC.

Bootloader ARM - Família iMX6



1. BootROM inicia e prepara a SRAM e USDHC.

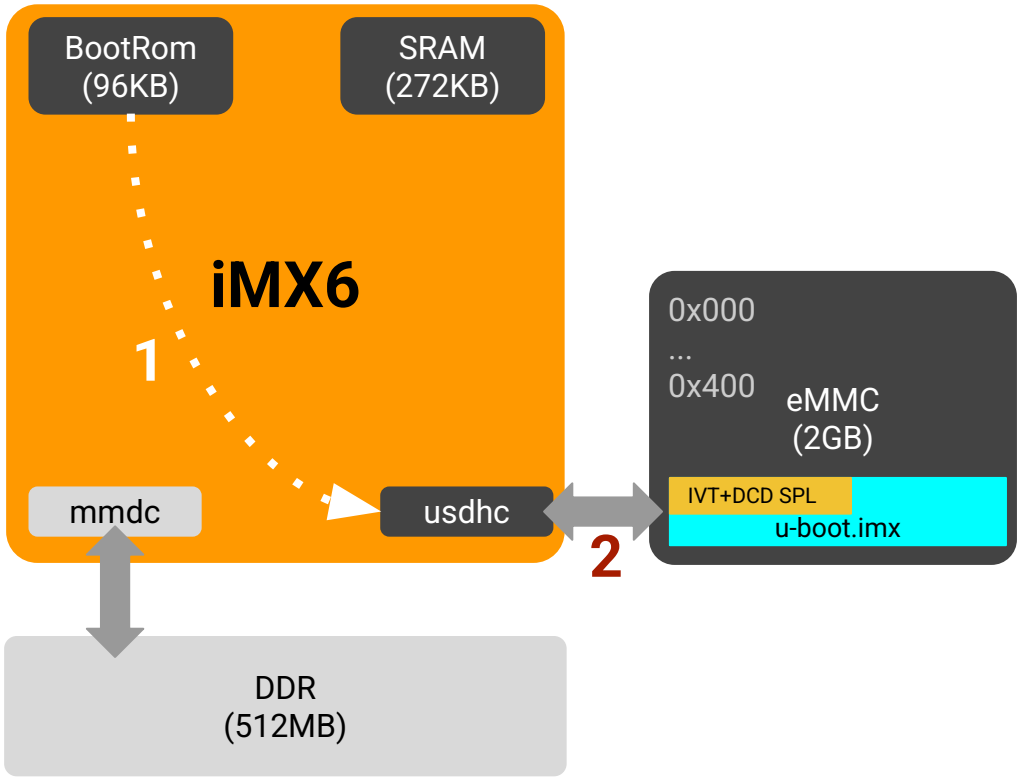
2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

IVT (Image Vector Table) (4KB)
- endereço de execução
- ponteiro para DCD
- tamanho da imagem (headers + DCD + SPL)

DCD (Device Configuration Data)
- configurar DDR
- configurar clocks básicos
- configurar IOMUX inicial

SPL (Secondary Program Loader)
- inicializa controladora mmdc(DDR)
- Carrega o U-Boot para Memória RAM

Bootloader ARM - Família iMX6



1. BootROM inicia e prepara a SRAM e USDHC.

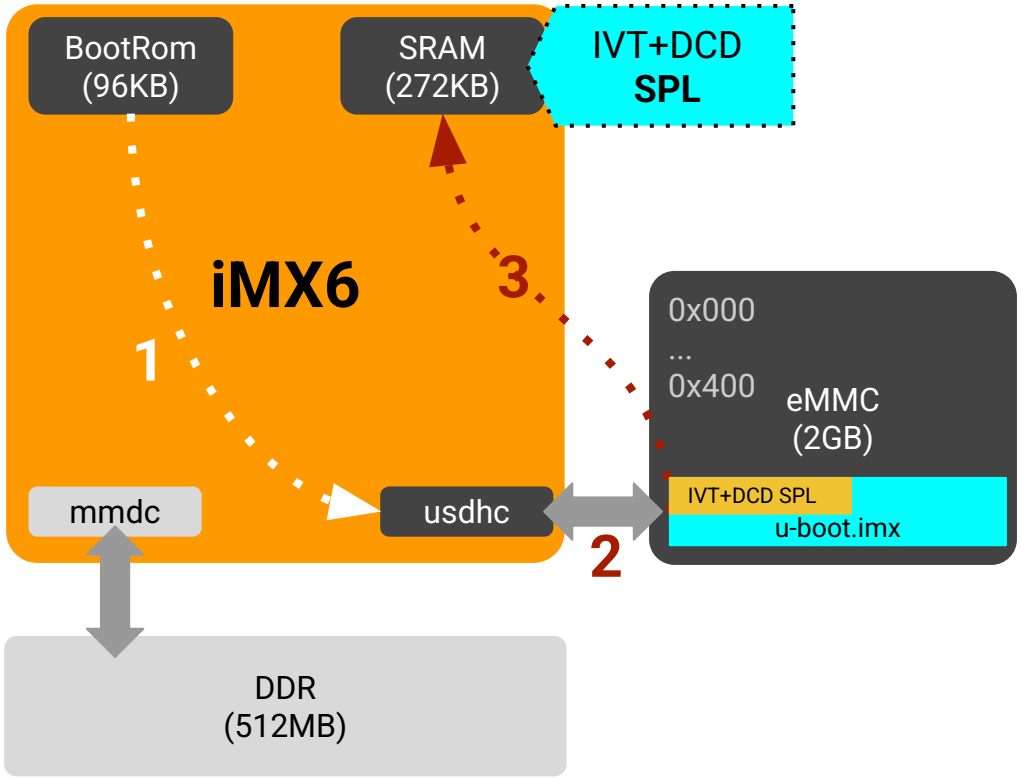
2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

IVT (Image Vector Table) (4KB)
- endereço de execução
- ponteiro para DCD
- tamanho da imagem (headers + DCD + SPL)

DCD (Device Configuration Data)
- configurar DDR
- configurar clocks básicos
- configurar IOMUX inicial

SPL (Secondary Program Loader)
- inicializa controladora mmdc(DDR)
- Carrega o U-Boot para Memória RAM

Bootloader ARM - Família iMX6

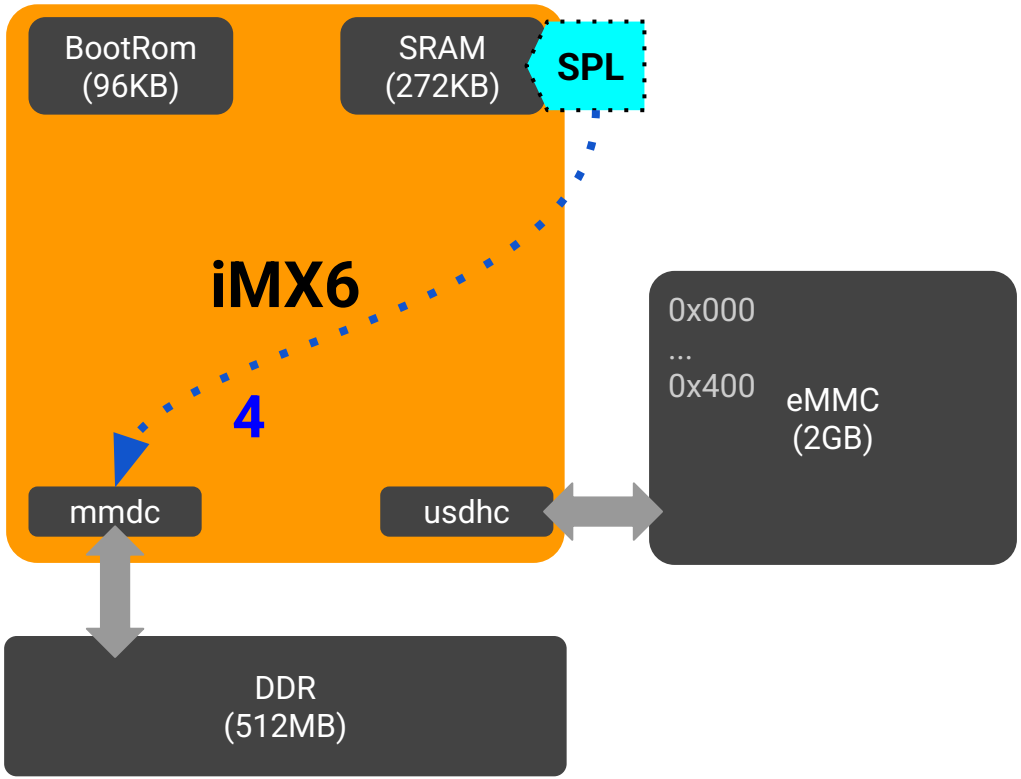


1. BootROM inicia e prepara a SRAM e USDHC

2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

3. Copia os dados para **SRAM** e executa o SPL.

Bootloader ARM - Família iMX6



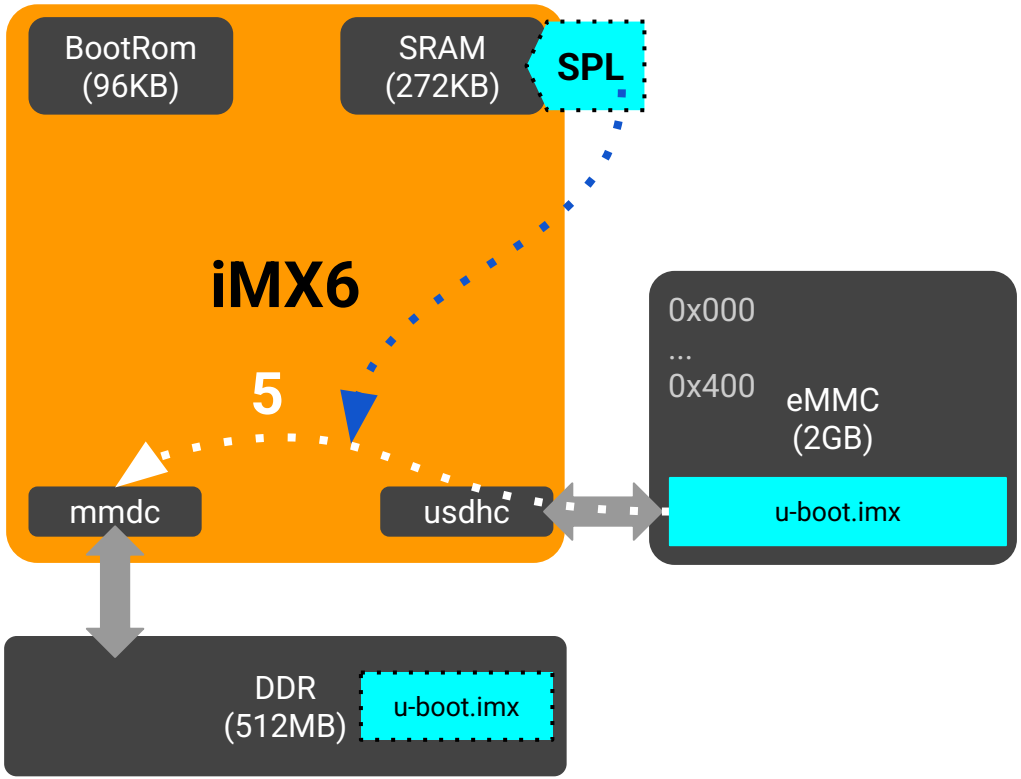
1. BootROM inicia e prepara a SRAM e USDHC.

2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

3. Copia os dados para **SRAM** e executa o SPL.

4. SPL inicializa Memória DDR via (Multi-Mode DDR Controller).

Bootloader ARM - Família iMX6



1. BootROM inicia e prepara a SRAM e USDHC.

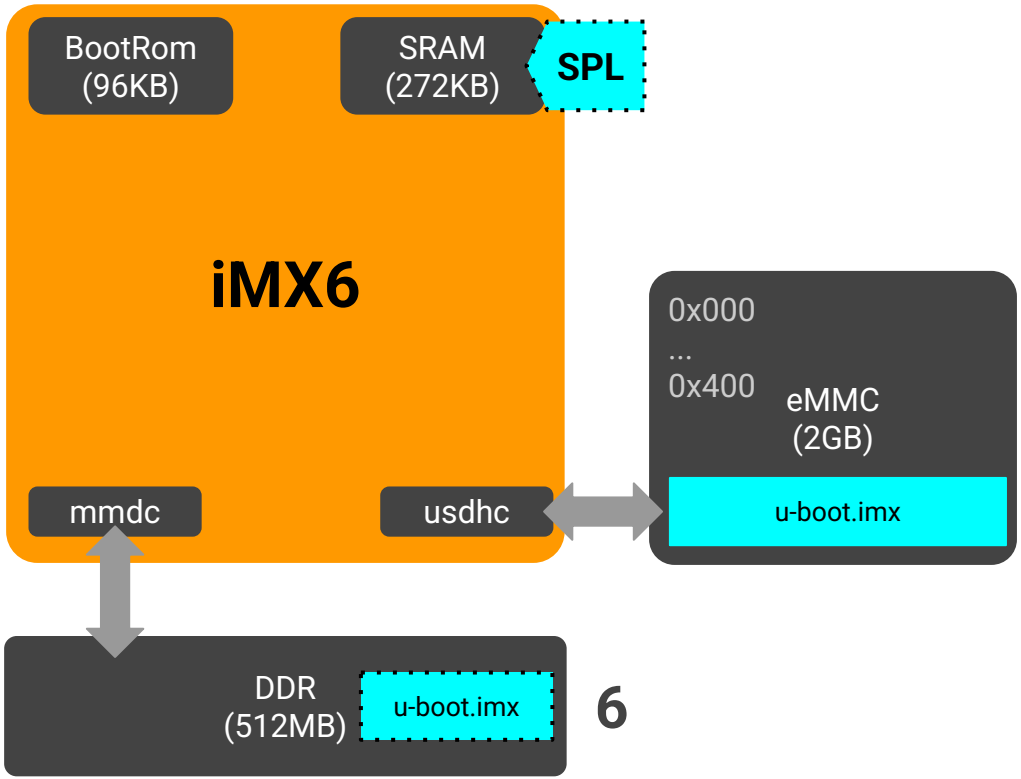
2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

3. Copia os dados para **SRAM** e executa o SPL.

4. SPL inicializa Memória DDR via (Multi-Mode DDR Controller).

5. SPL copia o bootloader(**uboot.imx**) para Memoria RAM.

Bootloader ARM - Família iMX6



1. BootROM inicia e prepara a SRAM e USDHC.

2. Lê o início do "Disco" e aplica um offset de 0x400(1KB) e carrega **IVT**, **DCD** e **SPL.**

3. Copia os dados para **SRAM** e executa o SPL.

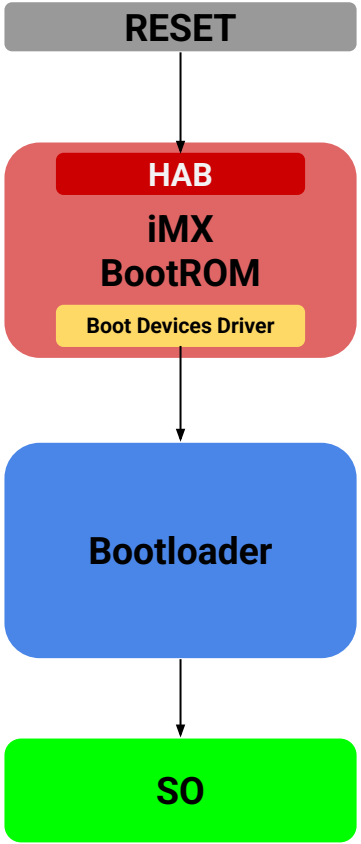
4. SPL inicializa Memória DDR via (Multi-Mode DDR Controller).

5. SPL copia o bootloader(**uboot.imx**) para Memoria RAM.

6. Executa o Bootloader

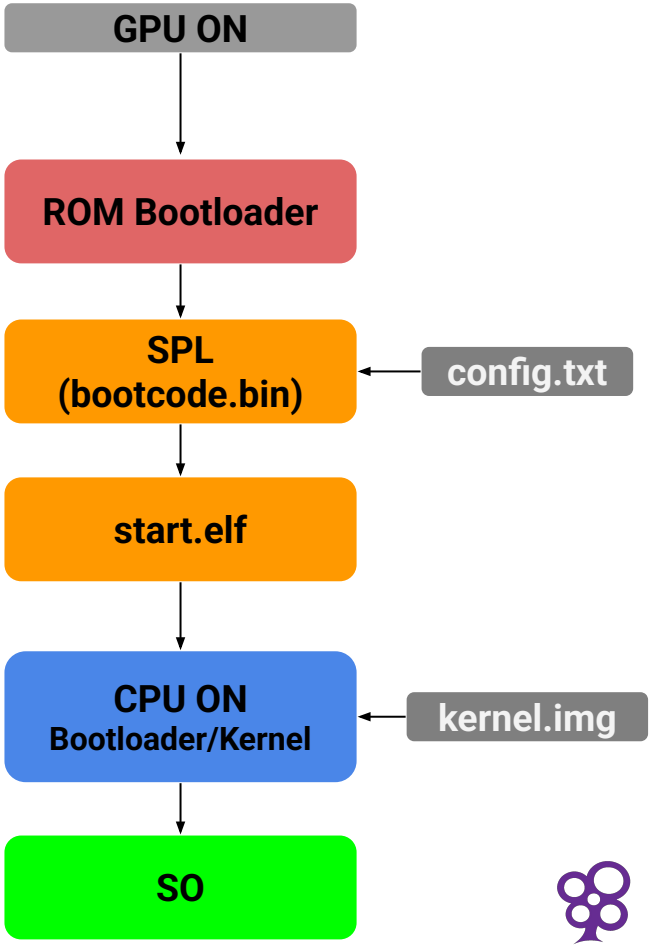
Bootloader ARM - Família iMX

Primeiro Estágio	**Inicialização básica**
Segundo Estágio	**Inicialização completa** - Opções de Boot - SPL



Bootloader ARM - RaspberryPI3

Primeiro Estágio	**Inicialização básica**
Segundo Estágio	**Inicialização básica** bootcode.bin(init. basic SoC) start.elf(firmwares)
Terceiro Estágio	**Inicializa o Kernel** ou Bootloader OpenSource

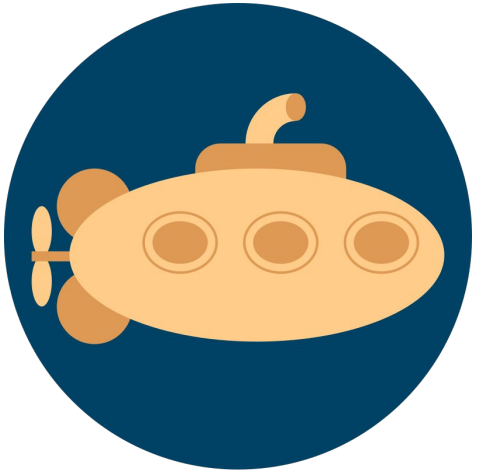


Bootloader ARM

- Entre diversas opções open-source de Bootloader para Embarcados, podemos destacar:
  - **U-Boot** - Universal Bootloader desenvolvido e mantido pela Denx, é o mais popular e utilizar em Embarcados sendo ARM, MIPS, PowerPC e até x86
  - **BareBox** - Desenvolvido pela Pengutronix, pouca adoção no mercado mas promete uma ótima alternativa ao U-Boot.

Bootloader - U-Boot

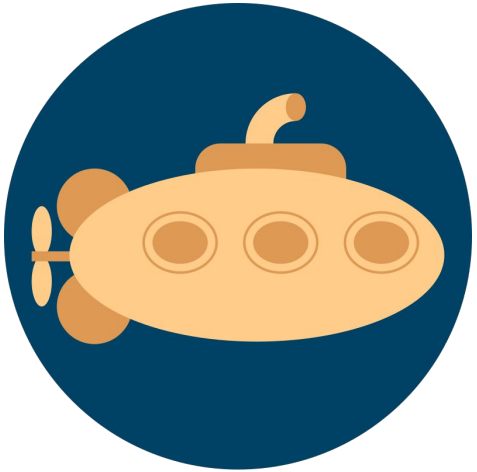
- Licença GPLv2(mesma do Kernel Linux)
- Mantido pela comunidade e empresas
- Projeto bem estruturado/organizado
- O repositório do projeto é disponibilizado em Git
- Cada 2~3 meses uma nova versão release é disponibilizada no formato YYYY.MM



U-Boot

Bootloader - U-Boot

- Prompt Interativo
- Exibe informações do hardware ao carregar
- Acesso via Serial Console
- Boot via: Rede(TFTP), MMC, USB, Flash SPI, NAND, etc
- Manipula variáveis ambiente e script interno
- Entender Sistemas de Arquivos EXT2/EXT3/EXT4, VFAT, UBIFS, etc
- Carrega o binário Kernel e passa parâmetros de inicialização



U-Boot

Bootloader - U-Boot

- Baixando o Bootloader U-Boot

```
$ git clone https://github.com/u-boot/u-boot.git

$ cd u-boot

u-boot$ git checkout v2026.01

u-boot$ git branch
* (HEAD detached at v2026.01)
  master
```

Bootloader - U-Boot

- Estrutura do projeto

```
u-boot$ ls
api          common     doc        fs         Licenses   README
arch         config.mk  drivers    include    MAINTAINERS scripts
board        configs    dts        Kbuild     Makefile   test
boot         COPYING    env        Kconfig    net        tools
cmd          disk       examples   lib        post
```

Bootloader - U-Boot

- Diretório **/configs** contém um ou várias arquivos de configuração para cada placa suportada
  - Ele define o tipo de CPU, os periféricos e sua configuração, e os recursos de U-Boot que devem ser suportados. Exemplos:

```
configs/am62x_beagleplay_a53_defconfig
configs/rpi_4_defconfig
```

Bootloader - U-Boot

- Antes de compilar o Bootloader U-Boot deve utilizar uma configuração visto no slide anterior, como exemplo realizando a configuração para uma **RaspberryPI4**:

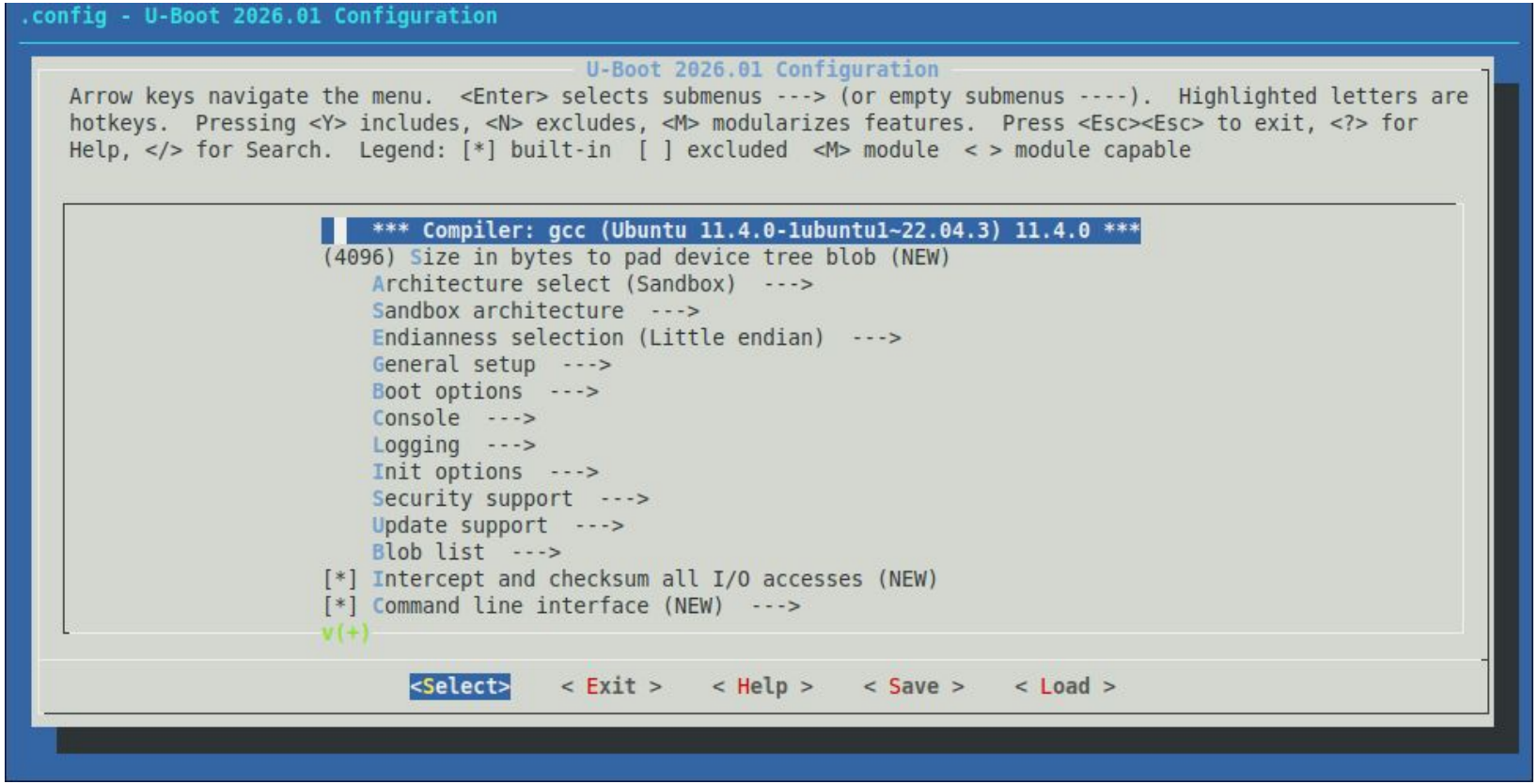
```
u-boot$ make rpi_4_defconfig
```

Bootloader - U-Boot <CONFIGURANDO>

- Um arquivo .config é gerado no mesmo diretório com as configurações padrões, você deverá alterar as configurações via menuconfig

```
u-boot$ make menuconfig
```

Bootloader - U-Boot <CONFIGURANDO>



Bootloader - U-Boot <COMPILANDO>

- Para compilar o U-Boot basta utilizar o comando **make** e fornecer o prefixo do toolchain configurado, como nosso compilado é **arm-unknown-linux-uclibcgnueabi-gcc** devemos utilizar **arm-unknown-linux-uclibcgnueabi-**:

```
u-boot$ make CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
...
  LDS     u-boot.lds
  LD      u-boot
  OBJCOPY u-boot.srec
  OBJCOPY u-boot-nodtb.bin
  COPY    u-boot.bin
  SYM     u-boot.sym
```

Bootloader - U-Boot <COMPILANDO>

- No final da compilação será gerada uma imagem do Bootloader U-Boot para ser gravada ou transferida para a placa de desenvolvimento, o arquivo gerado pode variar entre **u-boot, u-boot.bin, u-boot.img**.

```
u-boot$ file u-boot
u-boot: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
with debug_info, not stripped

u-boot$ ls -l u-boot
-rwxr-xr-x 1 b2open b2open 4417208 Jul 23 16:05 u-boot
```

Bootloader - LOAD_ADDRESS e ENTRY_POINT

- Duas variáveis importantes no carregamento e execução do Bootloader

  LOAD_ADDRESS - Endereço da memoria RAM onde o Bootloader u-boot será copiado

  ENTRY_POINT - Endereço que será chamado para executar o Bootloader

```
LOAD_ADDRESS  = 0x17800000
ENTRY_POINT   = 0x17819000    (Ex: ignora os primeiros 100 KB)
```

Bootloader - Carregando e Executando o Kernel

- O bootloader carrega o Kernel e o Device-Tree com comando **load** para um endereço na Memória RAM.

Descrição	Variável	RAM Address
Kernel Linux	kernel_addr_r	0x95c00000
Device-Tree Blob	fdt_addr_r	0x9d400000
RamDisk	ramdisk_addr_r	0x9d500000

Tabela acima baseado no Módulo **Toradex Colibri iMX8X**:

include/configs/colibri-imx8x.h

Bootloader - Carregando e Executando o Kernel

- Comando **bootz**(ARM 32-bit) e **booti**(ARM 64-bit) para executar o Kernel carregado

```
# Load kernel from SD card to RAM
fatload mmc 0:1 ${kernel_addr_r} zImage

# Load device tree
fatload mmc 0:1 ${fdt_addr_r} myboard.dtb

# Boot the kernel
bootz ${kernel_addr_r} - ${fdt_addr_r}
```

Bootloader - Carregando e Executando o Kernel

- Exemplo com Módulo **Toradex Colibri iMX8X**:

```
Colibri iMX8X # fatload mmc 0:1 ${kernel_addr_r} Image.gz
8318135 bytes read in 201 ms (39.5 MiB/s)

Colibri iMX8X # fatload mmc 0:1 ${fdt_addr_r}  imx8dx-colibri-aster.dtb
120592 bytes read in 6 ms (19.2 MiB/s)

Colibri iMX8X # booti ${kernel_addr_r} - ${fdt_addr_r}
   Uncompressing Kernel Image to 0
## Flattened Device Tree blob at 9d400000
   Booting using the fdt blob at 0x9d400000
Working FDT set to 9d400000
   Loading Device Tree to 00000000bd652000, end 00000000bd67270f ... OK

Starting kernel ...
```

Bootloader - Análise o bootcmd

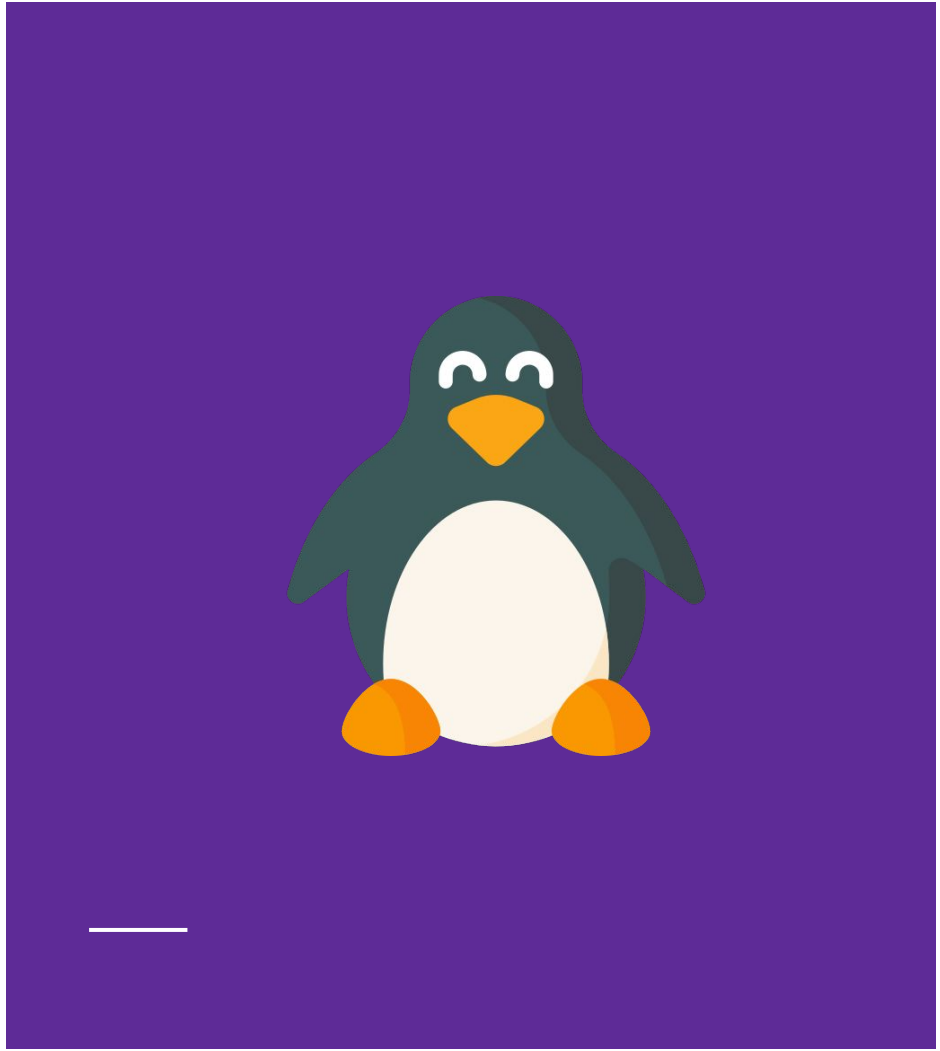
- **DICA:** Quer saber a sequência de inicialização, a ordem e de onde é carregado o kernel e device-tree? Análise a variável **bootcmd**!

```
Colibri iMX8X # printenv bootcmd
bootcmd=run distro_bootcmd

Colibri iMX8X # printenv distro_bootcmd
distro_bootcmd=for target in ${boot_targets}; do run bootcmd_${target}; done

Colibri iMX8X # printenv boot_targets
boot_targets=mmc1 mmc0 usb0 dhcp

Colibri iMX8X # printenv bootcmd_mmc1
bootcmd_mmc0=devnum=1; run mmc_boot
```

Laboratório

- Configurando e compilando o U-Boot para ARM
- Gravando U-Boot na Placa
- Prática Console Bootloader U-Boot

Kernel Linux



Um pouco de história

1991 - A origem de tudo 📅

"I'm doing a (free) operating system – just a hobby, won't be big and professional like GNU..."

- Linus Torvalds, 25 de agosto de 1991, em um post no grupo de discussão **comp.os.minix**

O e-mail histórico 📧

- Linus anunciou o Linux ao mundo com essa mensagem icônica. Inicialmente, ele desenvolvia o kernel como um hobby, inspirado no MINIX (sistema educacional de **Andrew Tanenbaum**).

Motivação

- Limitações do MINIX (software proprietário na época)
- Curiosidade técnica
- Interesse por sistemas Unix-like

Começou pequeno, mas com um grande diferencial

- Aberto para colaboração!

Um pouco de história

Crescimento exponencial

- Desenvolvedores do mundo inteiro começaram a contribuir
- O projeto GNU, iniciado por Richard Stallman, já tinha ferramentas (compilador, shell, etc.)
- Faltava apenas o **kernel** livre — e o Linux chegou para preencher essa lacuna!

Linux = Kernel

- O termo "Linux" refere-se ao **kernel**
- A combinação do kernel Linux com ferramentas GNU criou um sistema completo: o **GNU/Linux**

Hoje

- Roda desde supercomputadores até geladeiras
- É a base de Android, servidores, embarcados e IoT

Filosofia

- Liberdade de uso, modificação e redistribuição
- Comunidade colaborativa e global

Principais Recursos do Kernel Linux

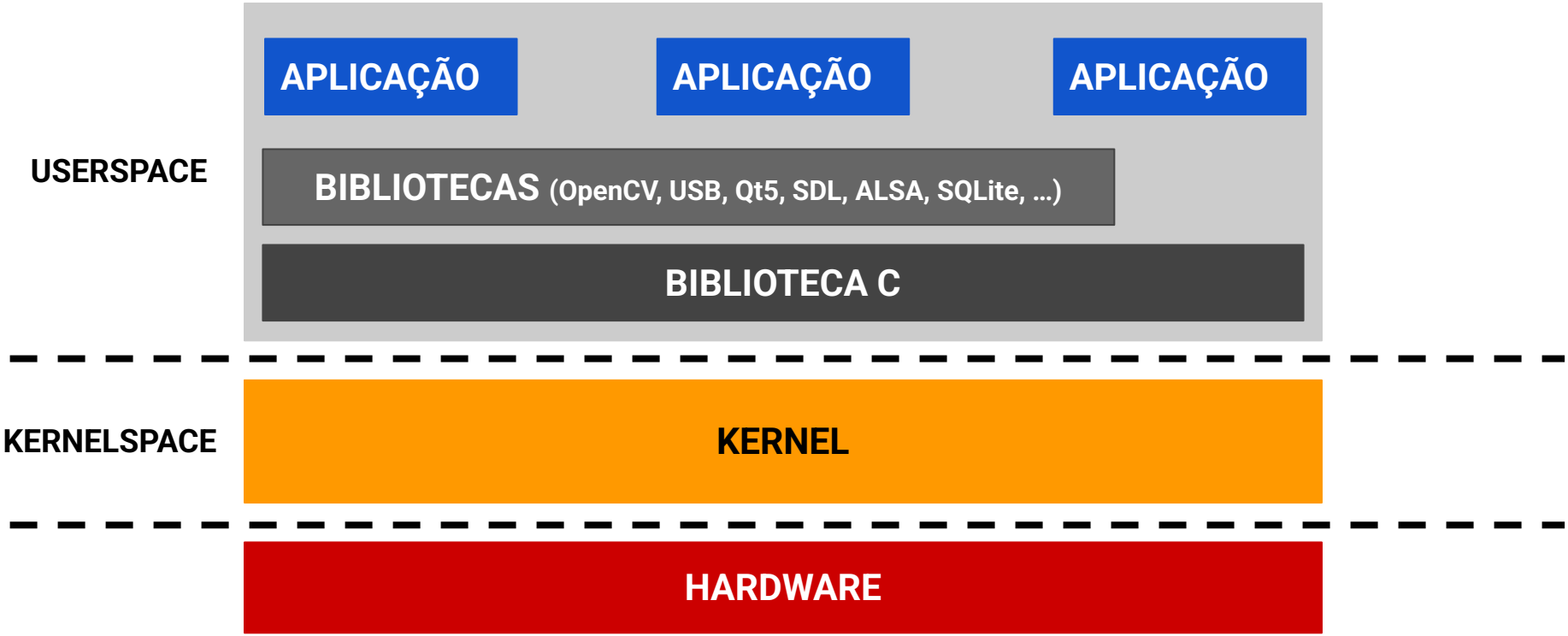
- Portabilidade e suporte de hardware.
- Suportado pela maioria das arquiteturas (veja **arch/** no código-fonte).
- Escalabilidade. Pode ser executado em MainFrames como em Dispositivos Embarcados.
- Segurança. Seu código é revisado por muitos especialistas.
- Estabilidade e confiabilidade (pode funcionar por dias/meses sem reiniciar)

Principais Recursos do Kernel Linux

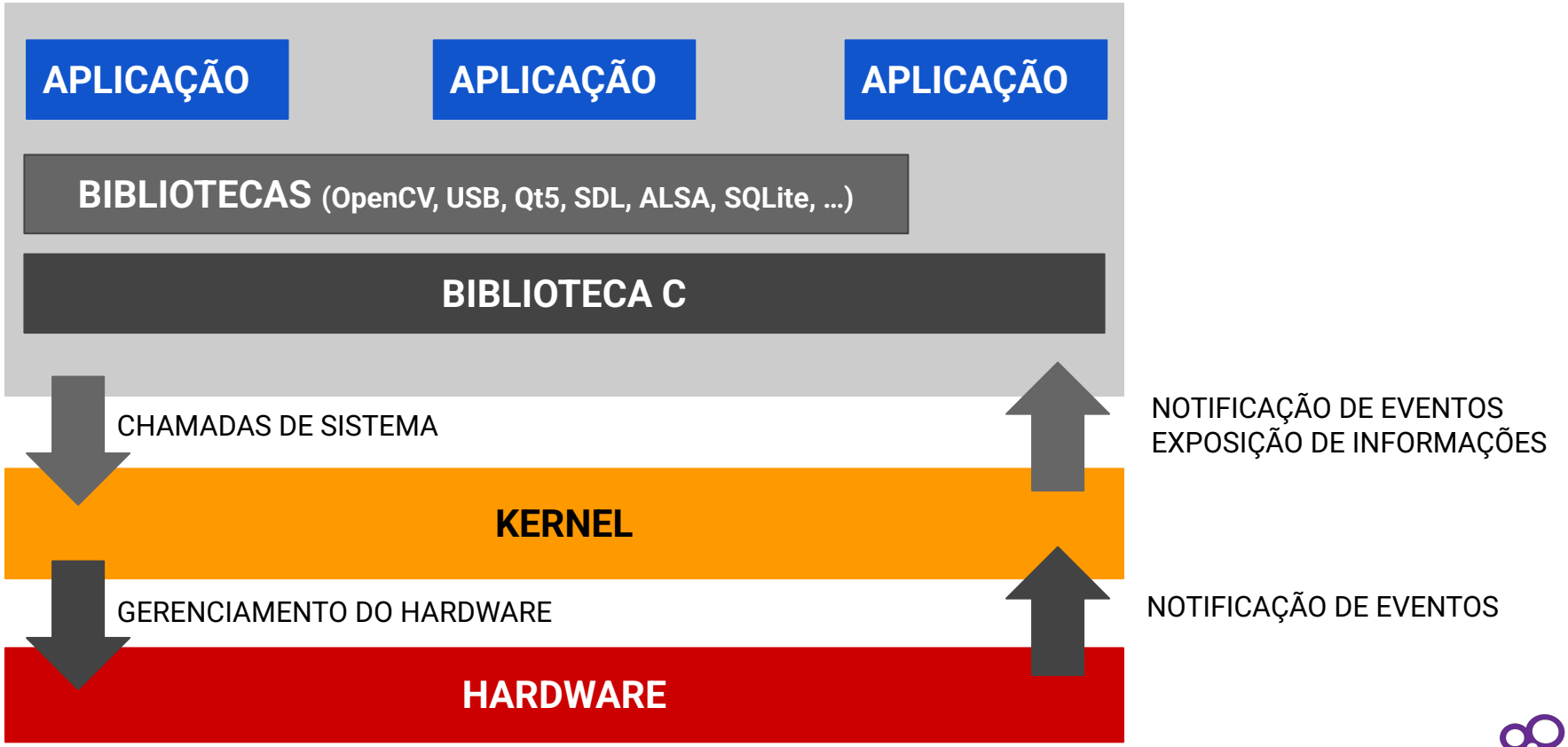
- Modularidade. Pode incluir apenas o que um sistema precisa, mesmo em tempo de execução (até mesmo o próprio kernel pode ser atualizado sem reiniciar - **kexec**)

- Todo código-fonte existente em sua maioria público, principal linguagem utilizada C, então pode aprender com o código existente, corrigir e implementar melhorias e novas funcionalidades

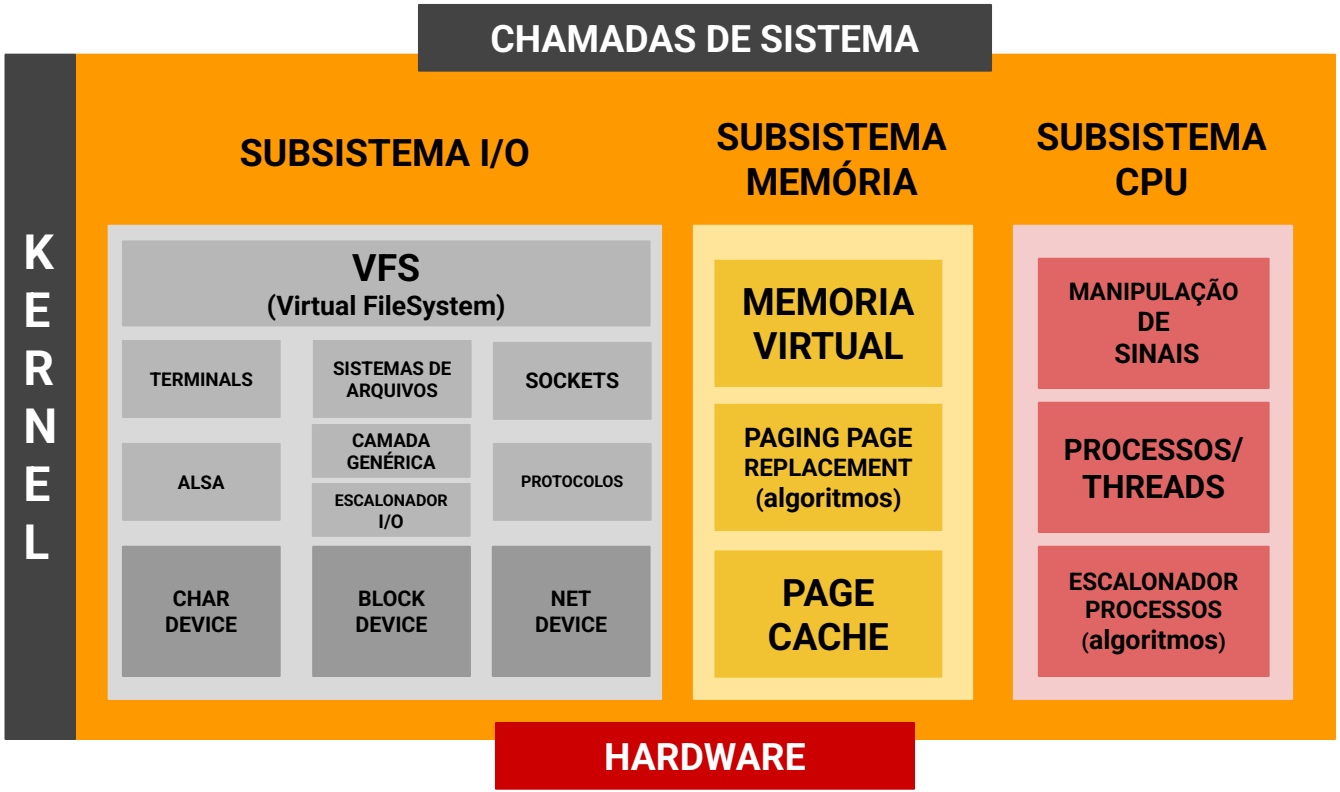
Por dentro do Kernel Linux



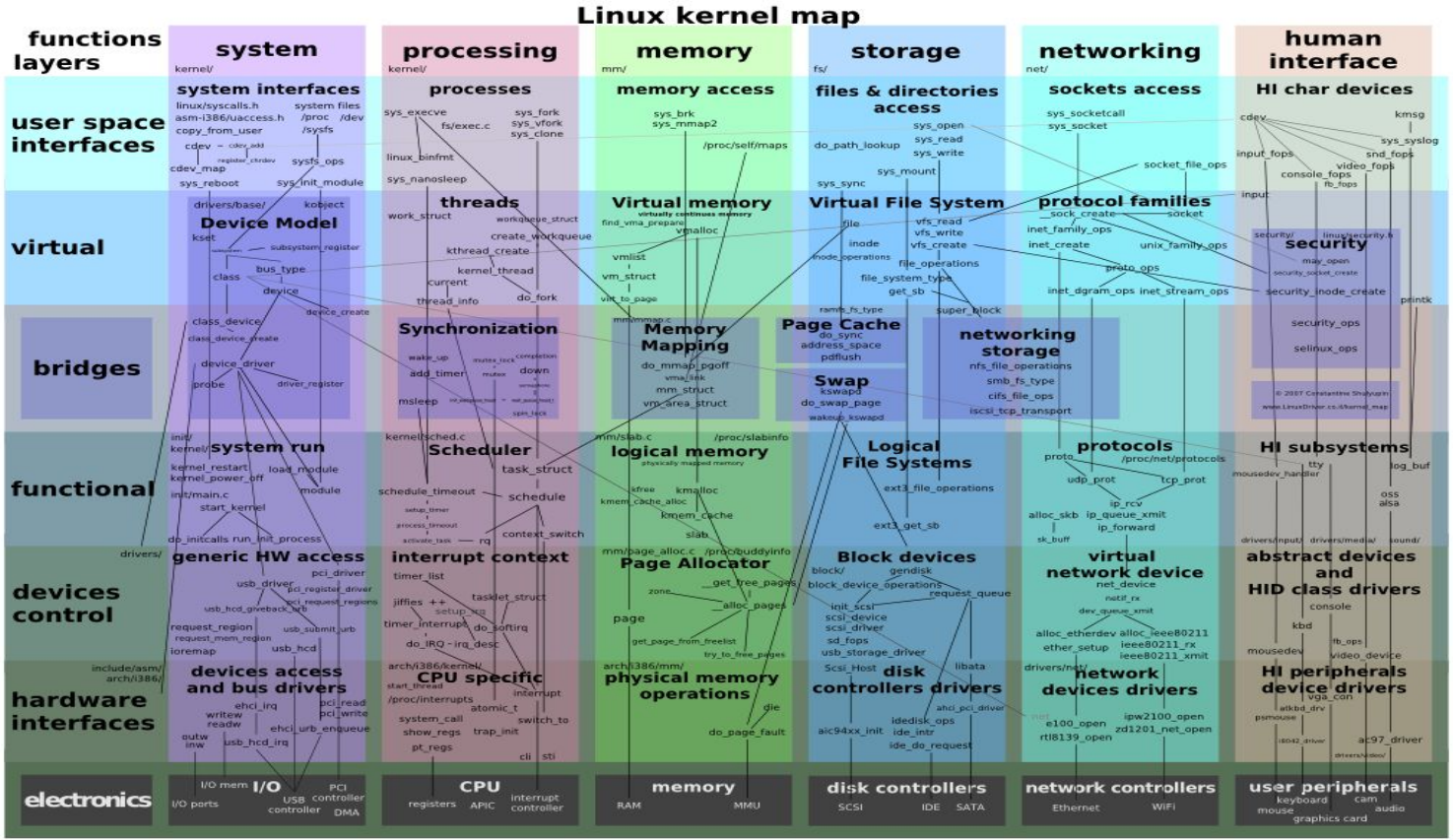
Por dentro do Kernel Linux



Por dentro do Kernel Linux



Por dentro do Kernel Linux



Referência: https://makelinux.github.io/kernel/map/

Por dentro do Kernel Linux

- Gerenciar todos os recursos de hardware: CPU, memória, I/O.

- Fornece um conjunto de APIs portáteis, de arquitetura e independentes de hardware para permitir que aplicativos e bibliotecas em userspace usem os recursos de hardware.

- O Kernel é responsável por multiplexar os acessos concorrente e compartilhados dos recursos de diferentes aplicações

Por dentro do Kernel Linux - VFS

- O Linux torna as informações do sistema e do kernel disponíveis no userspace por meio de pseudo sistemas de arquivos(VFS)

- No Linux praticamente podemos dizer que tudo é representado por um arquivo, mas com VFS podemos dizer pseudo-arquivos, com isso podemos manipular como um arquivo utilizando as chamadas **open**, **read**, **write**, **close**, **seek**, etc.

Por dentro do Kernel Linux - VFS

- Os dois principais VFS são:
  - **procfs** - Geralmente montado em **/proc**: Informações relacionadas ao Sistema Operacional (processos, parâmetros, gerenciamento de memória ...)
  - **sysfs** - Geralmente montado em **/sys**: Representação do sistema como uma árvore de dispositivos conectados por barramentos. Informações coletadas pelas estruturas do kernel que gerenciam esses dispositivos

Por dentro do Kernel Linux - LICENÇA

- Todo o código-fonte do kernel Linux é software livre e liberado sob a licença GPLv2.

Isso significa que:

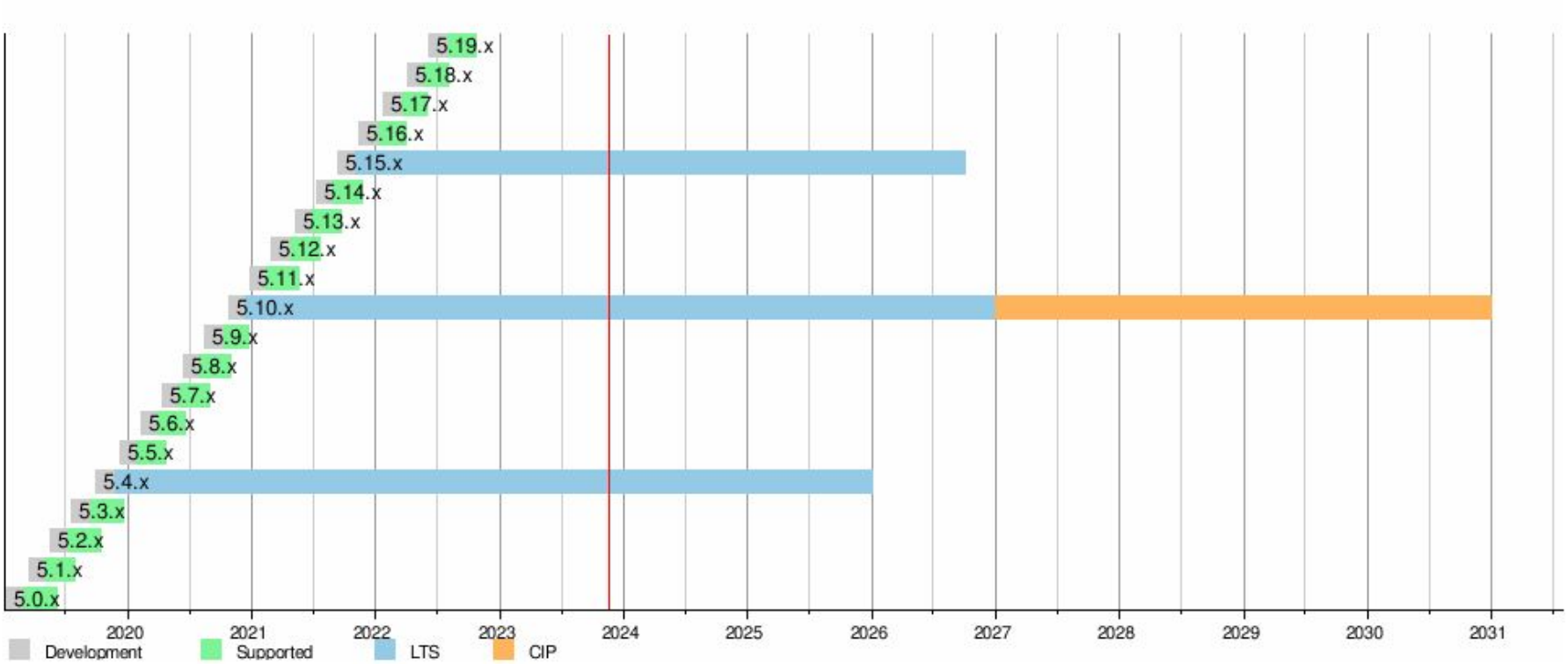
- Quando você receber ou comprar um equipamento com Linux embarcado, você tem o direito de requisitar os fontes, alterá-los e distribuí-los!
- Quando você produzir um equipamento com Linux embarcado, você precisa liberar os fontes do kernel sob as mesmas condições, sem restrições.

Por dentro do Kernel Linux - VERSIONAMENTO

- Até 2003, havia um novo branch de versão estável do Linux a cada 2 ou 3 anos (2.0, 2.2, 2.4). Novos ramos de desenvolvimento levaram 2-3 anos para se tornarem estáveis (era muito lento!).
- Desde 2003, há uma nova versão estável do Linux a cada 10 semanas:

Versões 2.6 (dezembro de 2003)	**a**	**2.6.39 (maio de 2011)**
Versões 3.0 (julho de 2011)	**a**	**3.19 (fevereiro de 2015)**
Versões 4.0 (abril de 2015)	**a**	**4.20 (dezembro de 2018)**
Versão 5.0 (março de 2019)		**Versão 6.0 (outubro de 2022)**

Por dentro do Kernel Linux - VERSIONAMENTO



Referência: https://en.wikipedia.org/wiki/Linux_kernel_version_history#Releases_5.x.y

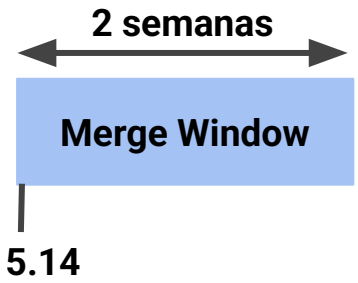
Por dentro do Kernel Linux - VERSIONAMENTO

- Recursos são adicionados ao kernel de forma progressiva. Desde 2003.

- Processo de desenvolvimento a cada 3 meses aproximadamente, até lançar o release final X.Y.

- O kernel Linux segue um **modelo de desenvolvimento cíclico** muito bem definido, geralmente de **aproximadamente 9 a 10 semanas**.

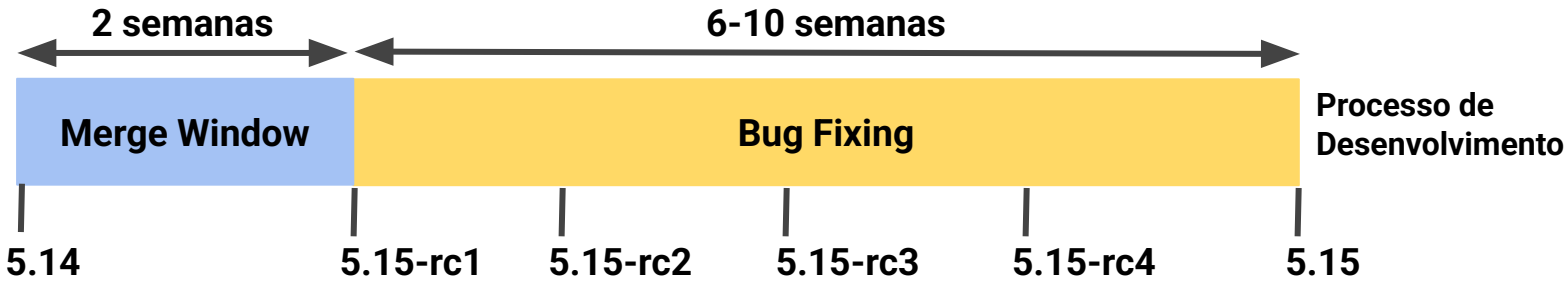
Por dentro do Kernel Linux - VERSIONAMENTO



🔁 Merge Window (Janela de Integração)

- ➔ **Duração:** 2 semanas
- ➔ **Começa imediatamente após** o lançamento de uma nova versão estável do kernel.
- ➔ Durante esse período, **Linus Torvalds** (mantenedor principal) aceita grandes mudanças, novos recursos e grandes atualizações vindas das **árvores dos mantenedores**.
- ➔ Essas mudanças são integradas à árvore principal (mainline) do kernel.
- ➔ Esta é a **única fase** onde novas funcionalidades são aceitas — depois disso, o foco é só em **estabilidade e correção de bugs**.

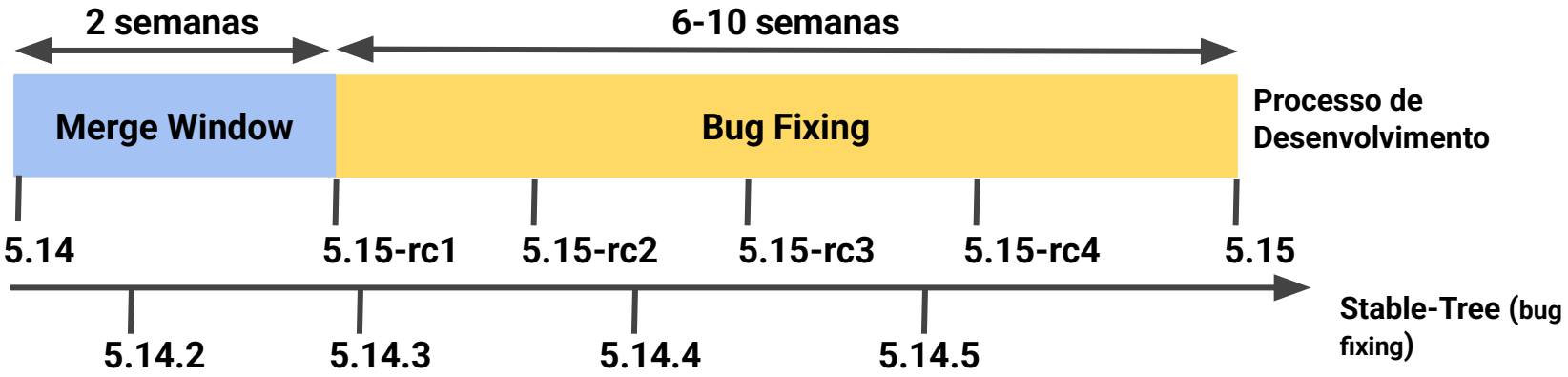
Por dentro do Kernel Linux - VERSIONAMENTO



🐞 Bug Fixing e Release Candidates (RCs)

- ➔ **Depois da Merge Window**, começam os **RCs (Release Candidates)**.
- ➔ Cada semana, uma nova versão RC é lançada: rc1, rc2, rc3, ..., até a versão final estável.
- ➔ **rc1**: contém todas as mudanças da Merge Window e
- ➔ **rc2 - rcN**: apenas correções de bugs, ajustes e pequenas melhorias.
- ➔ A comunidade testa intensamente essas versões para detectar e corrigir problemas.
- ➔ Após cerca de **7 a 8 RCs** (normalmente), uma nova **versão estável** é lançada
- ➔ Exemplo: `5.15-rc1` → `5.15-rc2` → ... → `5.15` (versão final)
- ➔ Após o lançamento, o ciclo recomeça com uma nova Merge Window para a próxima versão (`5.15`, por exemplo).

Por dentro do Kernel Linux - VERSIONAMENTO



🛠️ Manutenção das Versões Estáveis (Stable Trees)

Mesmo após o lançamento de uma versão estável, ela continua recebendo **correções de segurança e bugs críticos**.

Essas atualizações são feitas por mantenedores da **"stable tree"**, como Greg Kroah-Hartman. Exemplo:

- ➔ `5.15.1`, `5.15.2`... (correções incrementais para `5.15`)
- ➔ Essas atualizações seguem uma política rígida: **não introduzem novos recursos**, apenas correções.
- ➔ Após o lançamento, o ciclo recomeça com uma nova Merge Window para a próxima versão (`5.16`, por exemplo).

Por dentro do Kernel Linux - VERSIONAMENTO

- Para acompanhar as mudanças no kernel:
  - https://kernelnewbies.org/LinuxChanges
  - https://lwn.net/Kernel

Onde baixar o Kernel?

- As versões oficiais (mainline) do kernel Linux, conforme lançado por Linus Torvalds, estão disponíveis em: https://kernel.org
- As fontes do kernel estão disponíveis em https://kernel.org/pub/linux/kernel como tarballs completos (fontes completas do kernel) e patches (diferenças entre duas versões do kernel).
- Há opção de utilizar o repositório Git:
  - **git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git**
- Opção para visualizar o repositório via Web:
  - **https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/**

Onde baixar o Kernel?

- Exemplo baixando **tarball** e com **git**:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.57.tar.xz
```

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Onde baixar o Kernel?

- Comunidades e Fabricantes de hardware podem manter suas próprias versões do kernel:
  - Fabricantes de hardware fornecem suas próprias versões do kernel baseado em sua plataforma de referência (BSP)
  - Comunidade pode manter versões para arquiteturas especificar(ARM, MIPS, RISC-V) subsistemas(USB, network), Sistemas de Tempo-Real, etc

Onde baixar o Kernel?

- Kernel Toradex:

  https://git.toradex.com/cgit/linux-toradex.git/

- Kernel RaspberryPI:

  https://github.com/raspberrypi/linux

- Kernel Texas Instruments:

  https://git.ti.com/cgit/ti-linux-kernel/ti-linux-kernel/

- Kernel ST:

  https://github.com/STMicroelectronics/linux

Onde baixar o Kernel?

- O Kernel Linux da RaspberryPI - branch **rpi-5.4.y**
  - **66.405** arquivos
  - **27.938.153** linhas
  - **896.875.240** bytes
  - Tamanho total - **1.3G**
    - **arch/** - 162MB
    - **Documentation/** - 48MB
    - **drivers/** - 621MB
    - **fs/** - 41MB
    - **net/** - 32MB
    - **sound/** - 39MB

Onde baixar o Kernel?

- O Kernel Linux é grande!
- Um dos principais motivos é o gigantesco suporte a diversos drivers e arquiteturas
- No entanto, como o Kernel é modular e configurável o binário final é extremamente pequeno podendo ser de **1MB ~ 2MB**, casos com drivers de GPU, Codecs totalizando **5MB ~ 8MB**

Como configurar o Kernel?

- Modo Gráfico

```
$ make xconfig
```

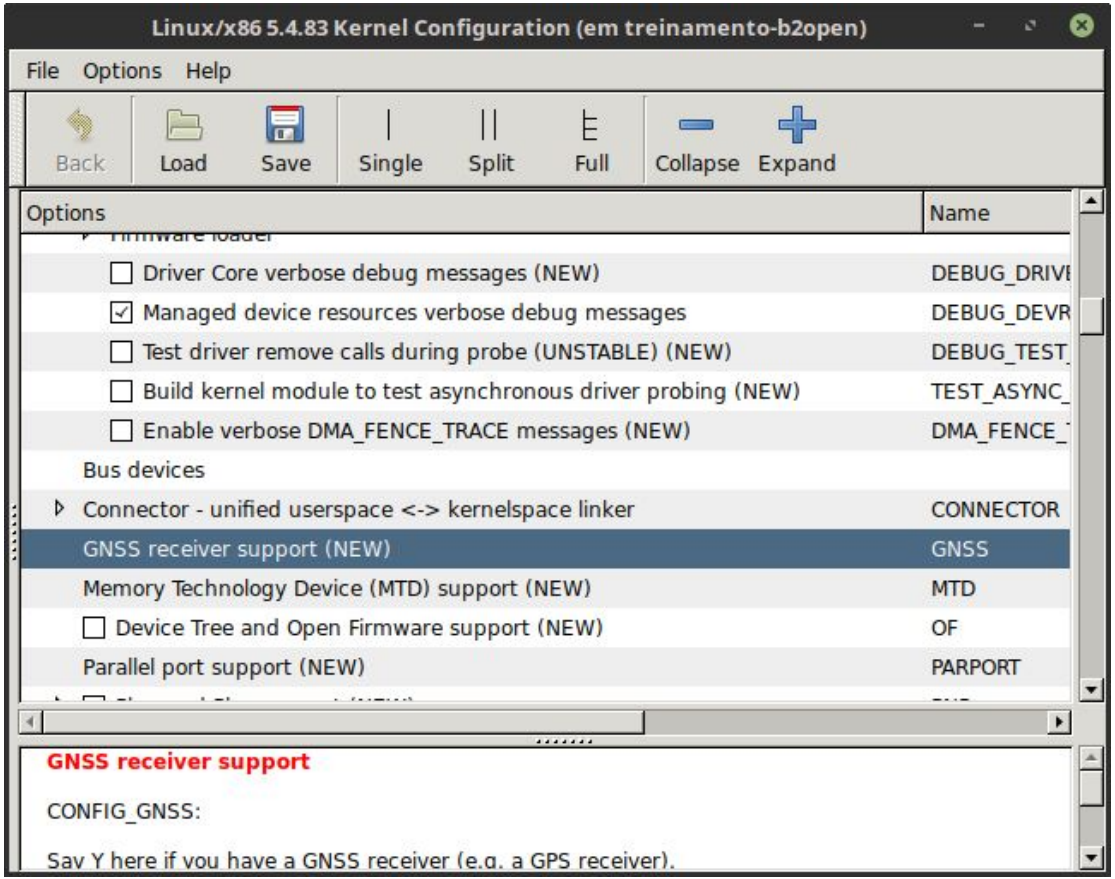
```
$ make gconfig
```

- Modo Texto

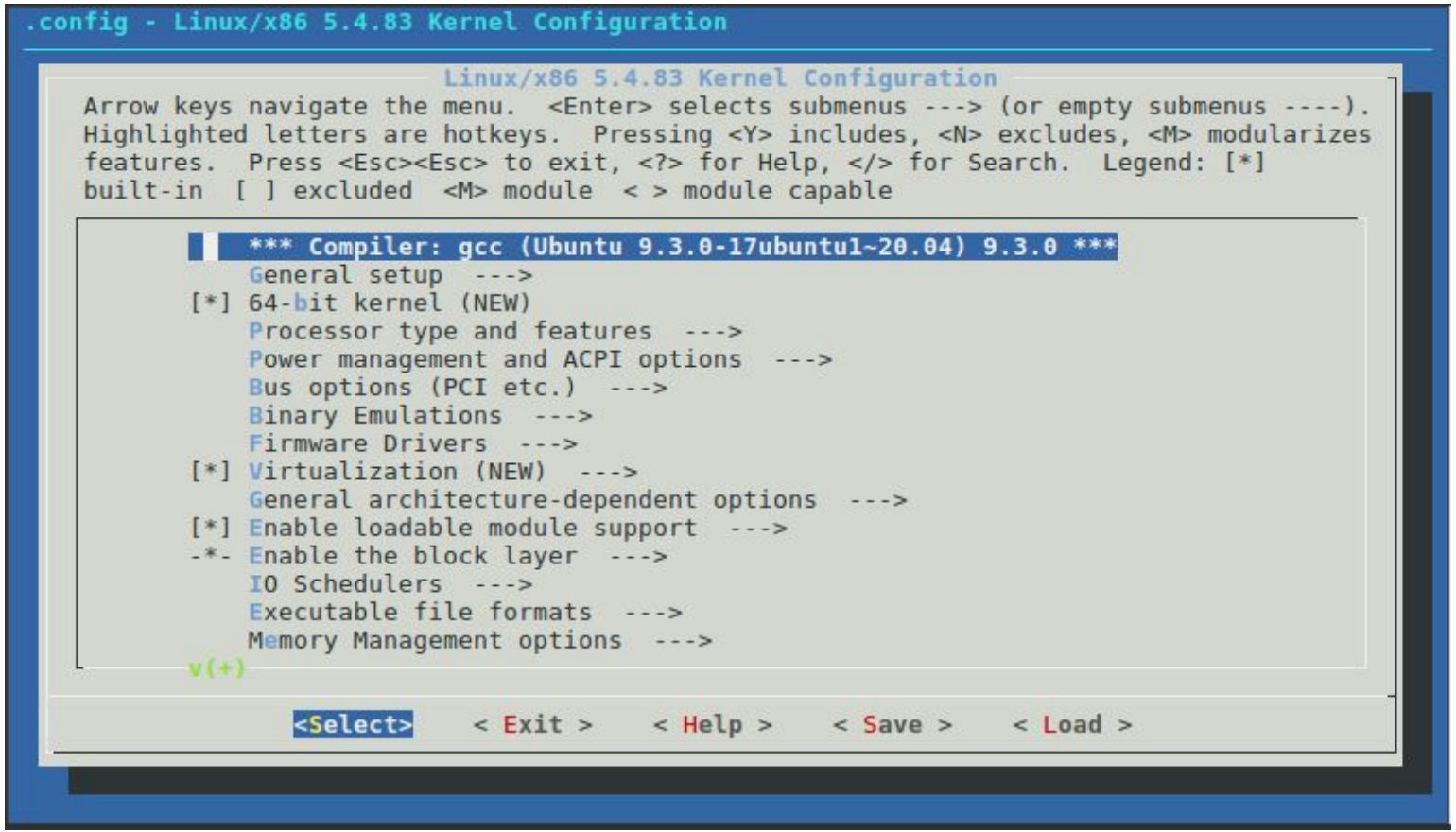
```
$ make menuconfig
```

```
$ make nconfig
```

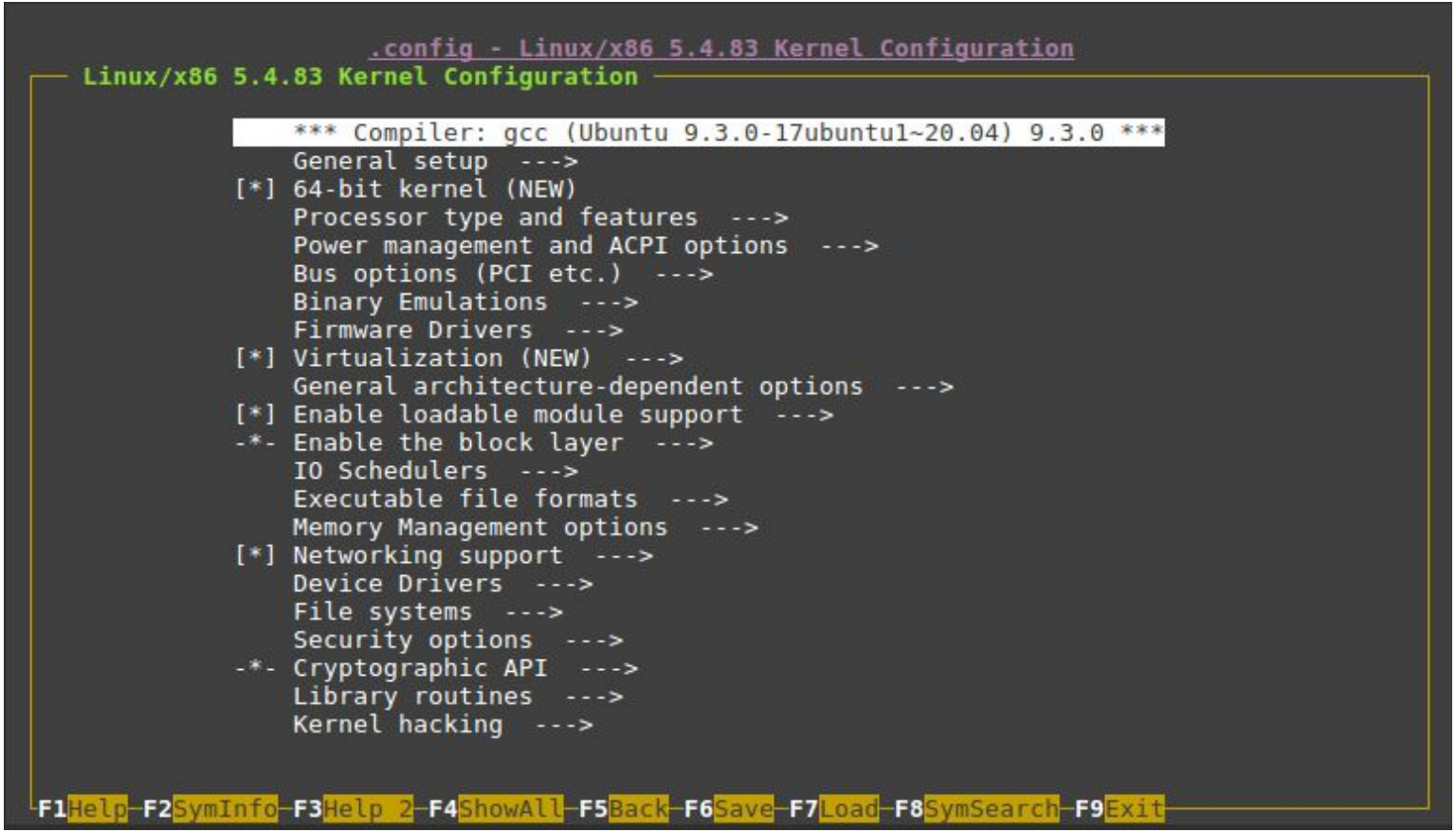
Como configurar o Kernel?



Como configurar o Kernel?



Como configurar o Kernel?



Como compilar o Kernel?

- Deve-se especificar a variável ARCH e CROSS_COMPILE
  - **ARCH** - Baseado na arquitetura alvo, veja **/arch**
  - **CROSS_COMPILE** - Especifica o prefixo do toolchain para (gcc, ld, strip, as, ...)

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

```
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi- ...
```

Configurações do Kernel

- As configurações do Kernel são armazenadas no arquivo .config
- Gerado e manipulado via **make menuconfig**, **make gconfig** ...
- Pode ser editado manualmente
- A configuração é baseado em CHAVE=VALOR
  - **CONFIG_EXFAT_FS=y** (Habilitado Built-in)
  - **CONFIG_BT_MRVL=m** (Habilitado como Módulo)
  - **CONFIG_SQUASHFS=n** (Desabilitado)

Kernel e Módulos

- O Kernel é gerado em um binário final com diversos recursos básicos e necessários para o boot completo

- Alguns recursos (drivers de dispositivo, sistemas de arquivos, etc.) podem, no entanto, ser compilados como módulos.
  - Esses são plug-ins que podem ser carregados/liberados dinamicamente para adicionar/remover recursos do kernel

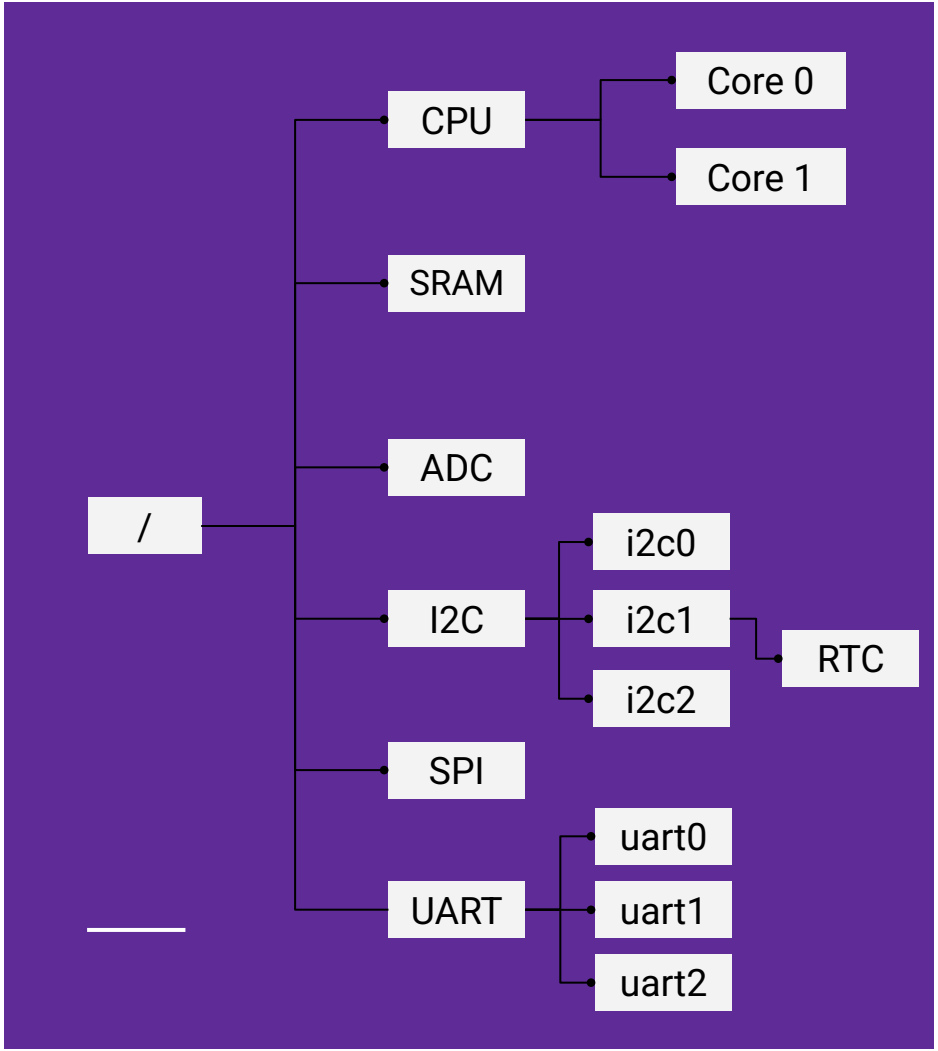
Resultado da compilação do Kernel

- O **vmlinux**, uma imagem descomprimida do kernel, em formato ELF, não possível realizar boot mas utilizado para depuração
- **arch/<arch>/boot/*Image**, a imagem final do kernel comprimida e que pode dar boot, alguns formatos:
  - bzImage para x86
  - zImage para ARM
- **arch/<arch>/boot/Image**, imagem kernel sem compressão que também pode inicializar
- **arch/<arch>/boot/dts/*.dtb**, resultado da compilação do device-tree

Resultado da compilação do Kernel

- Todos os módulos do Kernel(opções configuradas com =m), gerados com **.ko**(Kernel Object)
- Instalar os módulos requer o comando **make modules_install**
- **/lib/modules/<version>/**, diretório padrão de instalação dos módulos (.ko)

Device-Tree


/
CPU
Core 0
Core 1
SRAM
ADC
I2C
i2c0
i2c1
i2c2
RTC
SPI
UART
uart0
uart1
uart2

Device-tree x ACPI

Em **computadores/servidores** a inicialização conta com o auxílio da **BIOS/UEFI** e os periféricos são conectados ao processador principal por meio de um barramento(BUS), alguns barramentos suportam enumeração(discovery).

Desta forma o processador requisita ao barramento e o dispositivos retornam informações como tipo, fabricante, modelo e configurações.

O Sistema Operacional irá decidir qual driver irá utilizar para o dispositivos após receber estas informações.

Device-tree x ACPI

Em **sistemas embarcados**(ARM) não temos(no geral) BIOS/UEFI para auxiliar na inicialização, e contém barramentos que não há enumeração.

Com diversas plataformas ARM o Kernel Linux começou a ficar grande, complexo e muita duplicidade de código de diferentes fabricantes e a cada nova especificação mais código dentro do kernel. 

Alterar um simples endereço de um dispositivo na I2C teria que alterar o *platform device* dentro do kernel para cada plataforma ARM :/

Device-tree x ACPI

Em 2011, Linus Torvalds e alguns desenvolvedores do Kernel expressaram descontentamento com esse processo, estabelecendo um ultimato: eles se recusariam a aceitar novas implementações até que a ARM reorganizasse e estruturasse de maneira mais eficiente as implementações redundantes de plataformas, núcleos e drivers.

Precisava retirar a descrição de hardware do código em C e colocar em um dado/estrutura externa!

Referência: https://lwn.net/Articles/443510/

Device-tree x ACPI

Surge o **Device-Tree** que é derivado diretamente do modelo criado pelo Open Firmware.

- O Open Firmware (muito usado em sistemas PowerPC, como Macs antigos) já utilizava uma estrutura em árvore para descrever o hardware ao sistema operacional.
- Essa estrutura foi reaproveitada e adaptada para o Linux.

Inicialmente, o Linux (especialmente em ARM) usava "**board files**" - código C específico para cada placa.

Então a comunidade adotou o conceito do Open Firmware e criou o:

- **Flattened Device Tree** (**FDT**): versão binária (.dtb) usada pelo kernel

Device-tree x ACPI

- **Servidores/Computadores (x86) [ACPI]**

  Utiliza ACPI Table e possui enumeração/descoberta com barramentos como PCI, PCIe e USB

- **Sistemas Embarcados (ARM) [DEVICE TREE]**

  Utiliza Device Tree para inicialização e configuração dos dispositivos com barramentos como SPI e I2C

Device-tree - Resumindo

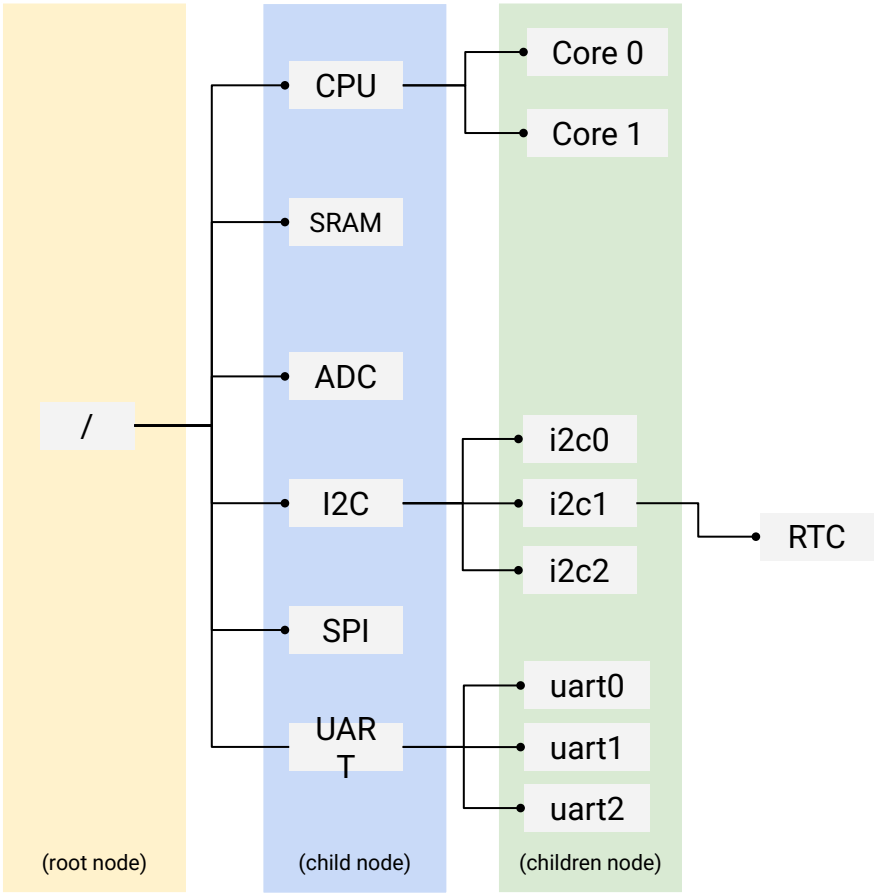
“ O **Device Tree** (DT) no contexto de sistemas embarcados e arquiteturas **ARM** é descrever a configuração de hardware de um sistema. Ele fornece uma representação hierárquica(Root Node, Child Node, Children Node) da configuração de hardware por meio de uma estrutura de árvore.”

O Kernel se encarrega de inicializar e/ou carregar o driver corretamente se necessário.

Device-tree <extensões>

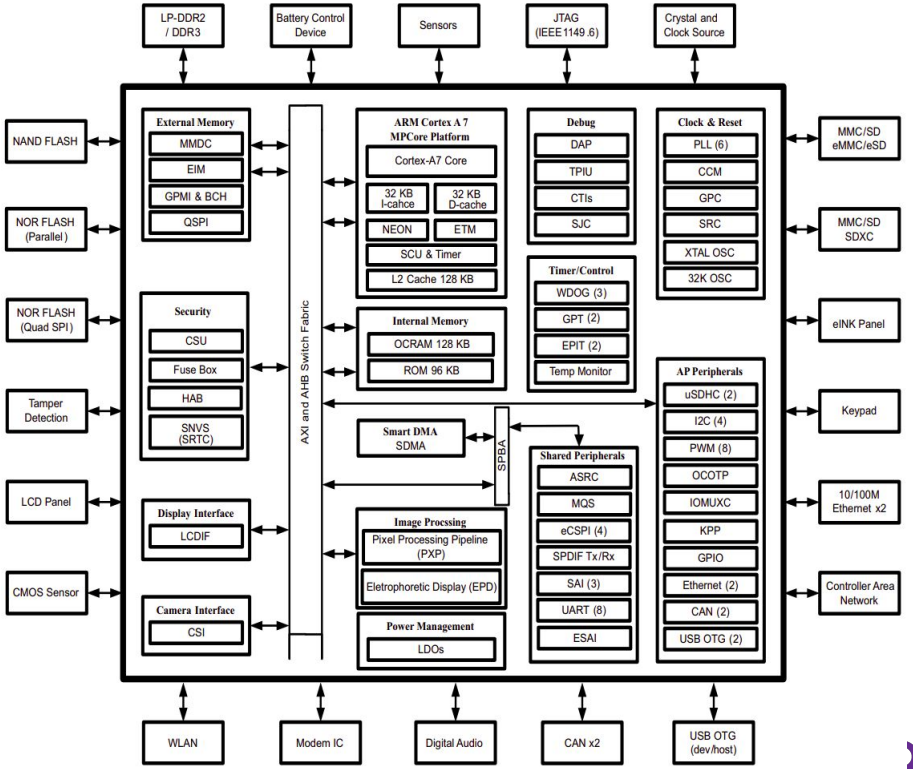
DTS	Device-Tree Source	Arquivo da descrição do hardware, o código-fonte
DTSI	Device-Tree Source Include	Arquivo de origem que contém definições comuns que podem ser incluídas em vários .dts
DTB	Device-Tree Blob	Arquivo resultado da codificação binária dos dados do .dts. O .dtb é o arquivo que o kernel Linux utiliza para obter as informações do hardware durante a inicialização
DTBO	Device-Tree Blob Overlay	Permite a personalização dinâmica do Device Tree, sem recompilar o original.
DTC	Device-Tree Compiler	Ferramenta que converte o arquivo .dts (source) em um formato binário chamado .dtb (binary).

Device-tree


/
CPU
SRAM
ADC
I2C
SPI
UAR
T
Core 0
Core 1
i2c0
i2c1
i2c2
RTC
uart0
uart1
uart2
(root node)
(child node)
(children node)

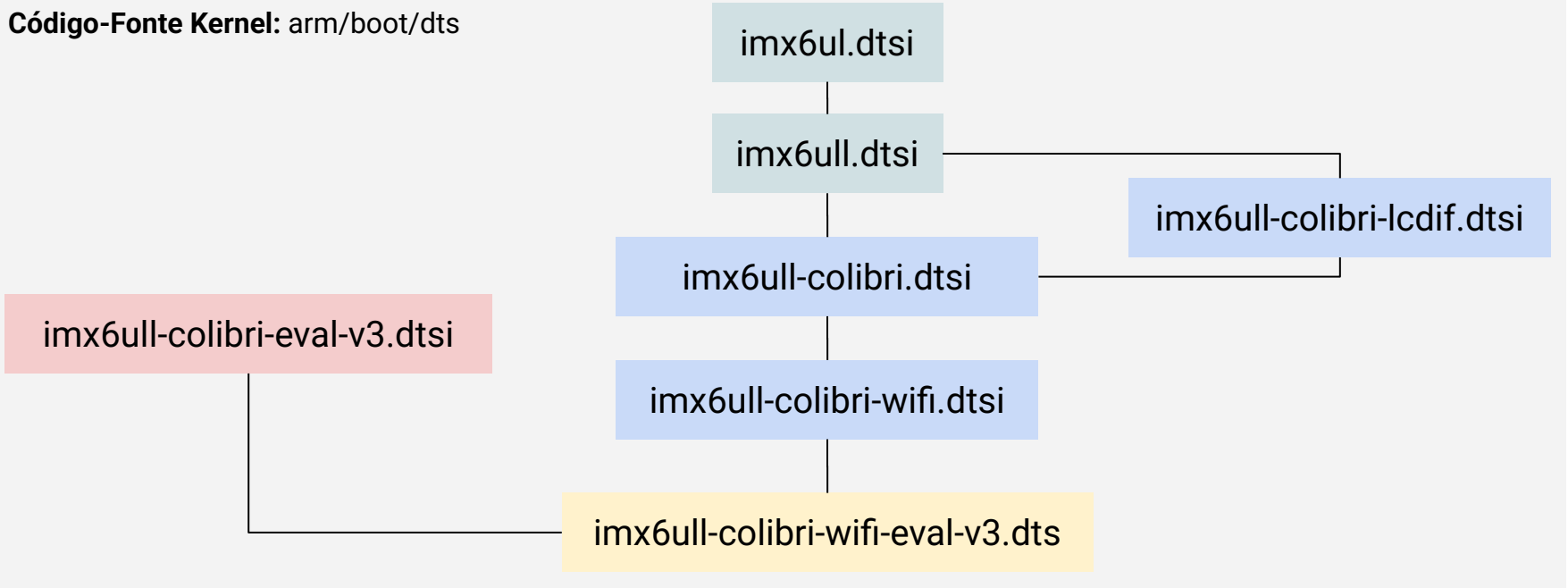
Device-tree

Para descrever um device-tree precisa de informações técnicas do Processador, SoC, Memória, Unidade de Armazenamento e CI's conectados nos periféricos.



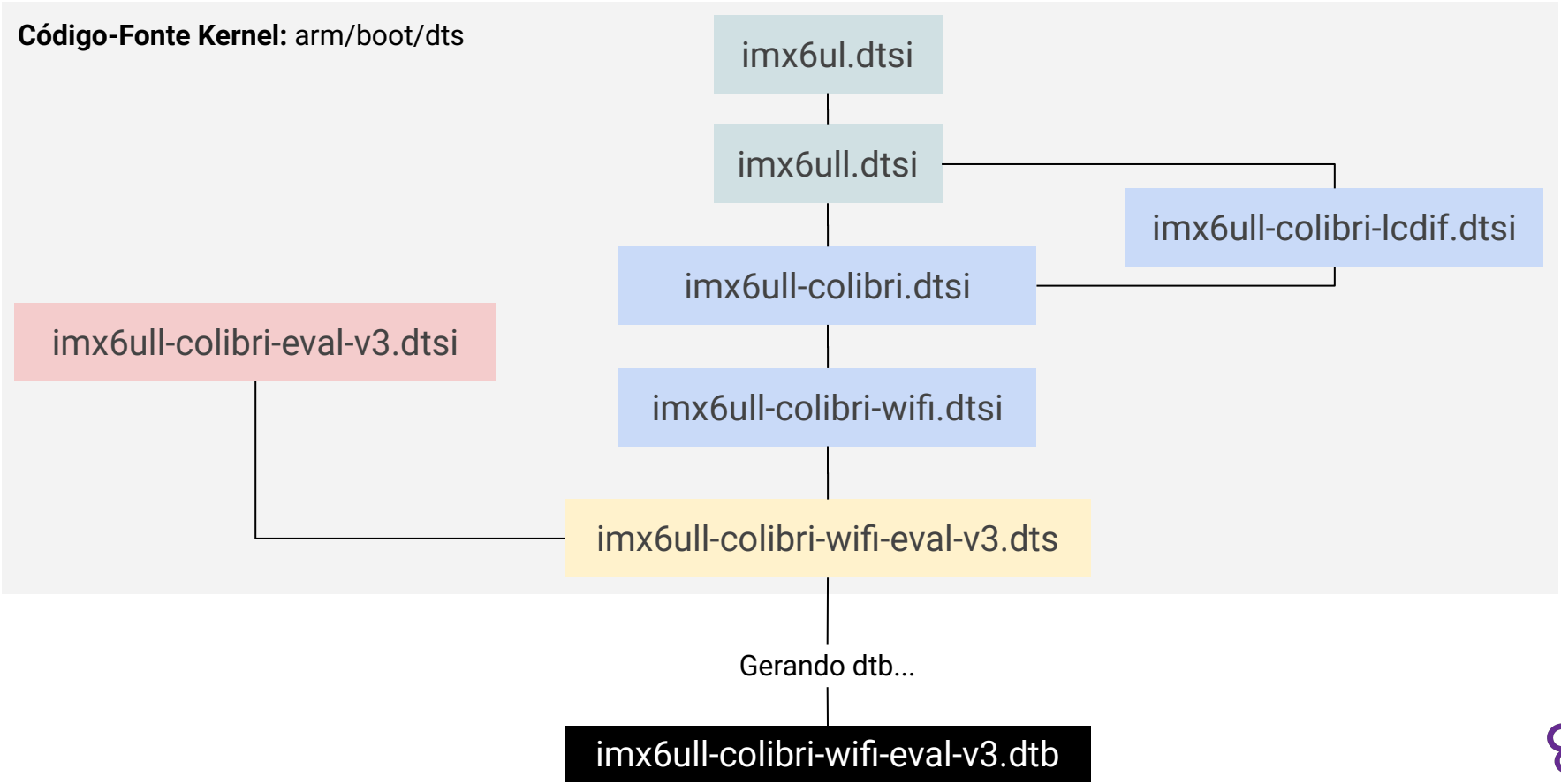
Device-tree <código-fonte e compilação>

Código-Fonte Kernel: arm/boot/dts

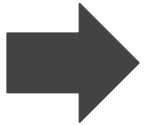


Device-tree <código-fonte e compilação>

Código-Fonte Kernel: arm/boot/dts



Device-tree <código-fonte e compilação>

imx6ull-colibri-wifi-eval-v3.dts  imx6ull-colibri-wifi-eval-v3.dtb

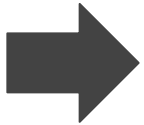
Para "compilar" manualmente o device-tree gerando o .dtb, acesse o diretório onde encontra-se o .dts:

cd arch/arm/boot/dts

Necessário para encontrar e resolver os includes .dtsi, comando **dtc** para compilar:

```
$ cd arch/arm/boot/dts

$ dtc  -I dts  -O dtb  -i .  imx6ull-colibri-wifi-eval-v3.dts  -o imx6ull-colibri-wifi-eval-v3.dtb
```

Device-tree <código-fonte e compilação>

imx6ull-colibri-wifi-eval-v3.dts  imx6ull-colibri-wifi-eval-v3.dtb

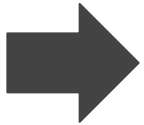
Se o .dts ou algum **.dtsi** inclui um **.h**(exemplo: imx6ul-pinfunc.h), o **dtc** retornará um erro como:

syntax error

Então precisa resolver .dts com .h com o comando **cpp** e em seguida utilizar **dtc**:

```
$ cpp -nostdinc -I include -I arch/arm/boot/dts -undef -x assembler-with-cpp
imx6ull-colibri-wifi-eval-v3.dts > imx6ull-colibri-wifi-eval-v3-temp.dts

$ dtc -I dts -O dtb -o imx6ull-colibri-wifi-eval-v3.dtb imx6ull-colibri-wifi-eval-v3-temp.dts
```

Device-tree <código-fonte e compilação>

imx6ull-colibri-wifi-eval-v3.dts  imx6ull-colibri-wifi-eval-v3.dtb

Se o .dts ou algum **.dtsi** inclui um **.h**(exemplo: imx6ul-pinfunc.h), o **dtc** retornará um erro como:

syntax error

Ou utilizar o make com o alvo **dtbs**, resolve estas inconsistências:

```
$ make ARCH=arm dtbs
```

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };

};
```

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };
};
```

Adicionar suporte ao RTC PCF8523

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };
};
```

Adicionar suporte ao RTC PCF8523

Documentation/devicetree/bindings/rtc

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };

};
```

```
pcf8523: rtc@68 {
        compatible = "nxp,pcf8523";
        reg = <0x68>;
};
```

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };

        pcf8523: rtc@68 {
                compatible = "nxp,pcf8523";
                reg = <0x68>;
        };

};
```

Device-tree <exemplo de modificação>

arch/arm/boot/dts/imx6ul.dts +30

```
aliases {
        ...
        i2c0 = &i2c1;
        i2c1 = &i2c2;
        i2c2 = &i2c3;
        i2c3 = &i2c4;
        mmc0 = &usdhc1;
        serial0 = &uart1;
        serial1 = &uart2;
        spi0 = &ecspi1;
        ...
};
```

arch/arm/boot/dts/imx6ul.dtsi +1008

```
i2c2: i2c@21a4000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
        reg = <0x021a4000 0x4000>;
        interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clks IMX6UL_CLK_I2C2>;
        status = "disabled";
};
```

arch/arm/boot/dts/imx6ull-colibri-eval-v3.dtsi +112

```
&i2c1 {
        status = "okay";

        /* M41T0M6 real time clock on carrier board */
        m41t0m6: rtc@68 {
                compatible = "st,m41t0";
                reg = <0x68>;
        };

        pcf8523: rtc@68 {
                compatible = "nxp,pcf8523";
                reg = <0x68>;
        };

};
```

Device-tree <exemplo de modificação>

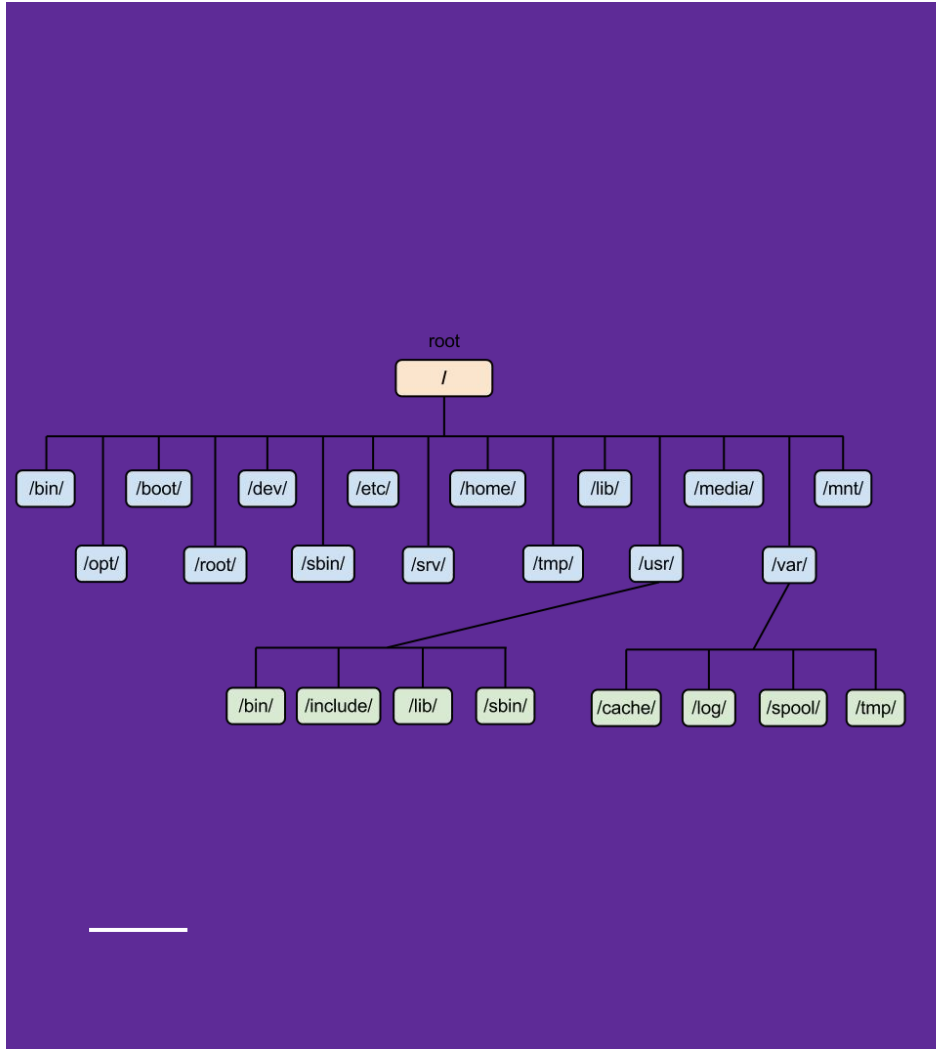
arch/arm/boot/dts/imx6ull-colibri-wifi-eval-v3.dts

```
/dts-v1/;

#include "imx6ull-colibri-wifi.dtsi"
#include "imx6ull-colibri-eval-v3.dtsi"

/ {
	model = "Toradex Colibri iMX6ULL 512MB on Colibri Evaluation Board V3";
	compatible = "toradex,colibri-imx6ull-wifi-eval", "fsl,imx6ull";
};

&i2c1 {
	status = "okay";

	pcf8523: rtc@68 {
		compatible = "nxp,pcf8523";
		reg = <0x68>;
	};
};
```

Laboratório

- Baixando, Configurando e compilando o Kernel para ARM
- Carregando o Kernel no Bootloader
- Primeiro Boot com o Kernel

RootFS


root
/
/bin/
/boot/
/dev/
/etc/
/home/
/lib/
/media/
/mnt/
/opt/
/root/
/sbin/
/srv/
/tmp/
/usr/
/var/
/bin/
/include/
/lib/
/sbin/
/cache/
/log/
/spool/
/tmp/

Root FileSystem

- Os sistemas de arquivos são usados para organizar dados em diretórios e arquivos em dispositivos de armazenamento ou na rede. Os diretórios e arquivos são organizados como uma hierarquia
- Em sistemas UNIX, aplicativos e usuários veem uma única hierarquia global de arquivos e diretórios.
- Sistemas de arquivos são montados em um local específico nesta hierarquia de diretórios.

Root FileSystem

- Quando um sistema de arquivos é montado(comando **mount**) em um diretório (chamado ponto de montagem), o conteúdo desse diretório reflete o conteúdo do dispositivo de armazenamento
- Quando o sistema de arquivos é desmontado(comando **umount**), o ponto de montagem fica vazio novamente.
- Isso permite que os softwares acessem arquivos e diretórios facilmente, independentemente de seu local de armazenamento exato.

Comando mount e umount

- **mount** permite montar sistemas de arquivos

```
$ sudo mount -t ext4 /dev/sdb1 /mnt/backup
```

  - **mount -t <type>** monta o dispositivo com o tipo especificado
  - **/dev/sdb1** é o dispositivo de armazenamento (HD, SSD, PenDrive, MicroSD, ...)
  - **/mnt/backup** é o diretório onde os arquivos do dispositivo de armazenamento serão montados
  - Diversas combinações de parâmetros são permitidas com mount para diversas funcionalidades, saiba mais sobre o mount

Comando mount e umount

- **umount** desmonta sistemas de arquivos

```
$ sudo umount /dev/sdb1
```

- Irá desmontar o ponto de montagem do dispositivo /dev/sdb1 tornando o diretório vazio
- Isso é necessário antes de reinicializar ou antes de desconectar um pendrive, porque o kernel armazena em cache as gravações na memória para aumentar o desempenho.

Sistema de Arquivo Root

- Um sistema de arquivos específico é montado na raiz da hierarquia, identificado por `/`
- Este sistema de arquivos é chamado de sistema de arquivos raiz
- Como os programas de mount e umount estão dentro do Sistema de Arquivos Raiz eles não fazem esta montagem, ficando a função ao Kernel resolver!

Sistema de Arquivo Root

- Como o sistema de arquivos raiz é o primeiro sistema de arquivos montado, ele não pode ser montado com o comando normal mount
- Ele é montado diretamente pelo kernel, de acordo com a opção **root=** kernel
- Quando nenhum sistema de arquivos raiz está disponível, o kernel lança um **Kernel Panic**:

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Sistema de Arquivo Root

- Ele pode ser montado de diferentes locais
  - Da partição de um disco rígido
  - Da partição de uma pendrive
  - Da partição de um cartão SD
  - Da partição de uma flash NAND
  - Da rede, usando o protocolo NFS(Network File System)

- O desenvolvedor deverá escolher o dispositivo correto do sistema e configurar no parâmetro **root=** do Kernel

Sistema de Arquivo Root - Montando RootFS

- Disco Rígido ou Pendrive

```
root=/dev/sdXY
```

X Número Identificação do Dispositivo
Y Número de Partição

- MicroSD ou eMMC

```
root=/dev/mmcblkXY
```

X Número Identificação do Dispositivo
Y Número de Partição

- Flash NAND (exemplo com UBIFS)

```
ubi.mtd=3 root=ubi0:rootfs
```

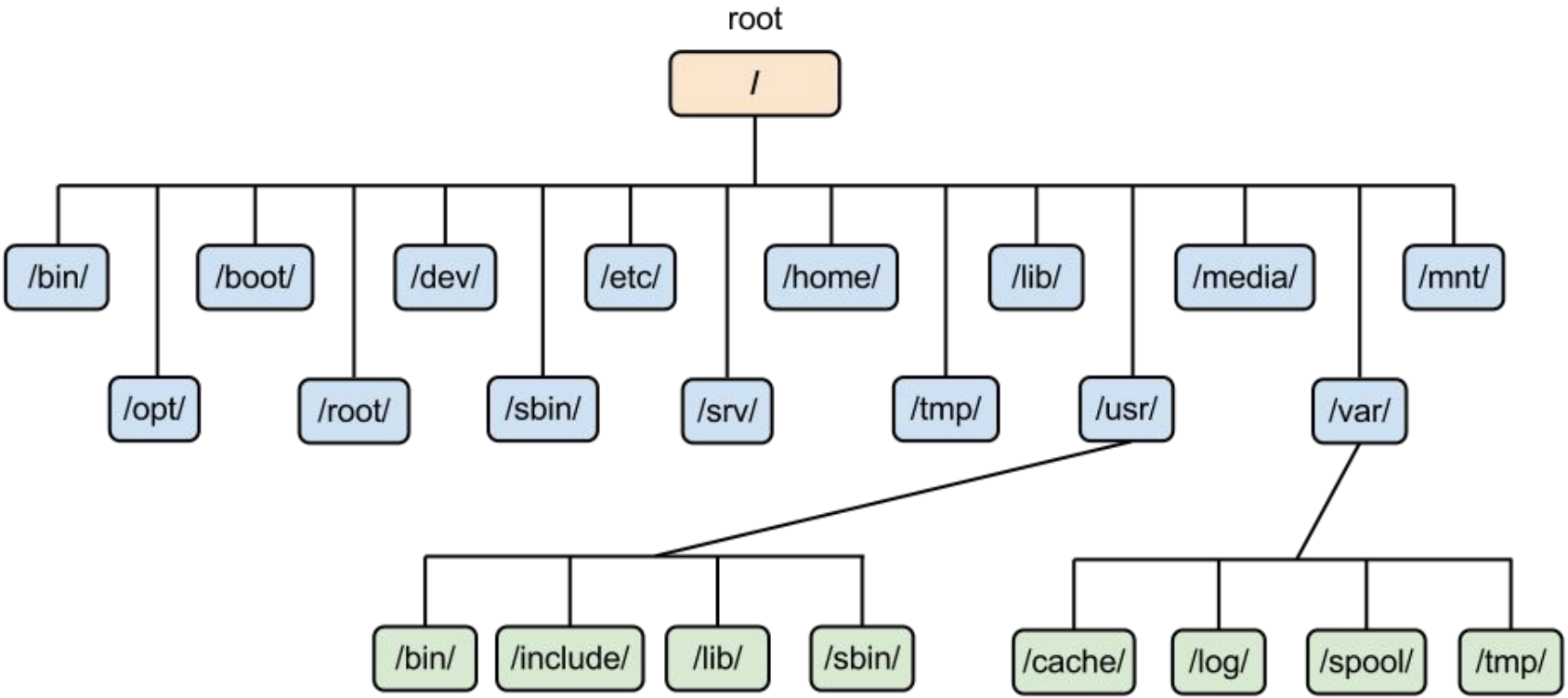
Sistema de Arquivo Root - InitramFS

- Também é possível inicializar o sistema com um sistema de arquivos na memória: **initramfs**
  - A partir de um arquivo CPIO compactado integrado à imagem do kernel
  - No momento da inicialização, este arquivo é extraído para o cache do Linux
  - É útil para dois casos:
    - **FastBoot** para um sistema muito pequeno
    - **OTA**(OverTheAir) - Uma etapa intermediária antes de mudar para um sistema de arquivos raiz real

Sistema de Arquivo Root - Organização

- A organização de um RootFS do Linux em termos de diretórios é bem definida pelo **FHS**(Filesystem Hierarchy Standard)
  - https: //wiki.linuxfoundation.org/lsb/fhs
- A maioria dos sistemas Linux está em conformidade com esta especificação
- Os aplicativos esperam esta organização
- Isso torna mais fácil para desenvolvedores e usuários, pois a organização do sistema de arquivos é semelhante em todos os sistemas

Sistema de Arquivo Root - Organização



Sistema de Arquivo Root - Organização

/bin	Programas Básicos do Sistema
/sbin	Programas de Administração do Sistema
/boot	Arquivos de Bootloader, Kernel e Device-Tree
/etc	Arquivos de Configuração do Sistema
/home	Diretórios dos usuários comuns do Sistema
/root	Diretório do usuário root
/lib	Bibliotecas Essenciais do Sistema e os Módulos do Kernel

Sistema de Arquivo Root - Organização

/mnt	**Diretório de Montagem dos Dispositivos**
/proc	**Diretório Virtual Informações Processos e Sistema - procfs**
/sys	**Diretório Virtual Controlado pelo Kernel e Módulos - sysfs**
/tmp	**Arquivos Temporários**
/usr/bin /usr/lib /usr/sbin	**Aplicações Básicas dos Usuários** **Bibliotecas dos Usuários** **Aplicações de Administração dos Usuários** *Unix System Resources*
/var	**Arquivos Variáveis (Temporários, Logs, Banco de Dados, ...)**

Sistema de Arquivo Root - Organização

/dev — **Onde todos os Dispositivos do Sistema são criados**

- Uma função interessante do kernel é permitir que os programas aplicativos acessem dispositivos de hardware como arquivos

- Internamente, o kernel identifica cada dispositivo por:
  - Type (Dispositivo de Caracteres ou Dispositivo de Bloco)
  - Major Number (normalmente a categoria do dispositivo)
  - Minor Number (normalmente o identificador do dispositivo)

Sistema de Arquivo Root - Organização

/dev — **Onde todos os Dispositivos do Sistema são criados**

- Exemplo de Dispositivos em um Sistema Linux

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda /dev/sda1 /dev/sda2 /dev/sdc1 /dev/zero
brw-rw---- 1  root disk   8,  0   2011-05-27 08:56 /dev/sda
brw-rw---- 1  root disk   8,  1   2011-05-27 08:56 /dev/sda1
brw-rw---- 1  root disk   8,  2   2011-05-27 08:56 /dev/sda2
brw-rw---- 1  root disk   8,  32 2011-05-27 08:56 /dev/sdc
crw-------   1  root root   4,  1   2011-05-27 08:57 /dev/tty1
crw-rw----  1 root dialout    4,  64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root   1,  5   2011-05-27 08:56 /dev/zero
```

Sistema de Arquivo Root - Organização

/dev — **Onde todos os Dispositivos do Sistema são criados**

- Exemplo de código C que usa a API de arquivo usual para gravar dados em uma porta serial

```
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd,"Hello",5);
close(fd);
```

Sistema de Arquivo Root - Criando Arquivos de Dispositivos

- Nas versões de Kernel < 2.6 e sistemas mais simples, os arquivos de dispositivo podem ser criados manualmente com o comando **mknod** (é necessário privilégios de root):

```
$ mknod /dev/<device>  [c|b]  MAJOR  MINOR
```

- Para sistemas mais complexos, existem mecanismos para adicionar e remover arquivos de dispositivo dinamicamente:
  - udev
  - mdev
  - devtmpfs

Sistema de Arquivo Root - Criando Arquivos de Dispositivos

devtmpfs

O Sistema de Arquivos Virtual devtmpfs é montado em **/dev** e contém todos os dispositivos registrados nas estruturas do kernel. A opção de configuração do kernel **CONFIG_DEVTMPFS_MOUNT** faz com que o kernel o monte automaticamente no momento da inicialização.

mdev

Uma versão minimalista do udev, disponibilizada pelo **Busybox**, mais adequado para uso em sistemas embarcados.
Pode ser executado em userspace e analisar **/sys** por novos dispositivos, necessario configurar **echo /sbin/mdev > /proc/sys/kernel/hotplug**

udev

O **udevd** é iniciado, ele “escuta” uevents do Kernel, que são enviados sempre que dispositivos são inseridos ou removidos. O **udevd** lê e analisa todas as regras encontradas em **/etc/udev/rules.d/** e *cria/modifica/remove* o dispositivo de acordo com as regras, API para integrar direto em aplicações.
Utilitário **udevadm** para monitorar eventos e regras

Sistema de Arquivo Virtual - ProcFS

- O sistema de arquivos virtual **procfs** existe desde o início do Linux
- Ele permite:
  - O kernel expor estatísticas sobre os processos em execução no sistema
  - O usuário pode ajustar em tempo de execução diversos parâmetros do sistema, gerenciamento de processos, gerenciamento de memória: Cada programa em execução possui uma entrada em **/proc/<PID>**
- Programas como **ps** e **top** não funciona sem o sistema de arquivos **procfs**
- Comando para montar /**proc**: **mount -t proc nodev /proc**
- Mais detalhes do procfs em Documentação ProcFS ou **man proc**

Sistema de Arquivo Virtual - ProcFS

- Alguns exemplos e informações úteis do **procfs**:

/proc/cmdline	Informações dos parâmetros passado para o Kernel no Boot
/proc/devices	Informações de todos os dispositivos reconhecidos pelo Kernel durante o boot
/proc/interrupts	Relaciona o número de interrupções por CPU por dispositivo de I/O, ele exibe o número de IRQ
/proc/filesystems	Todos os sistemas de arquivos atualmente suportados pelo kernel
/proc/<PID>/*	Diversas informações sobre o processo em execução, baseado no ID do Processo <PID>
/proc/uptime	Tempo em que o sistema está ligado

Sistema de Arquivo Virtual - SysFS

- Permite representar no userspace as informações que o kernel tem dos barramentos, dispositivos e drivers do sistema.

- É útil para vários aplicativos de userspace que precisam listar e consultar o hardware disponível, por exemplo **udev** ou **mdev**.

- Todos os aplicativos que usam **sysfs** esperam que ele seja montado no diretório **/sys**.

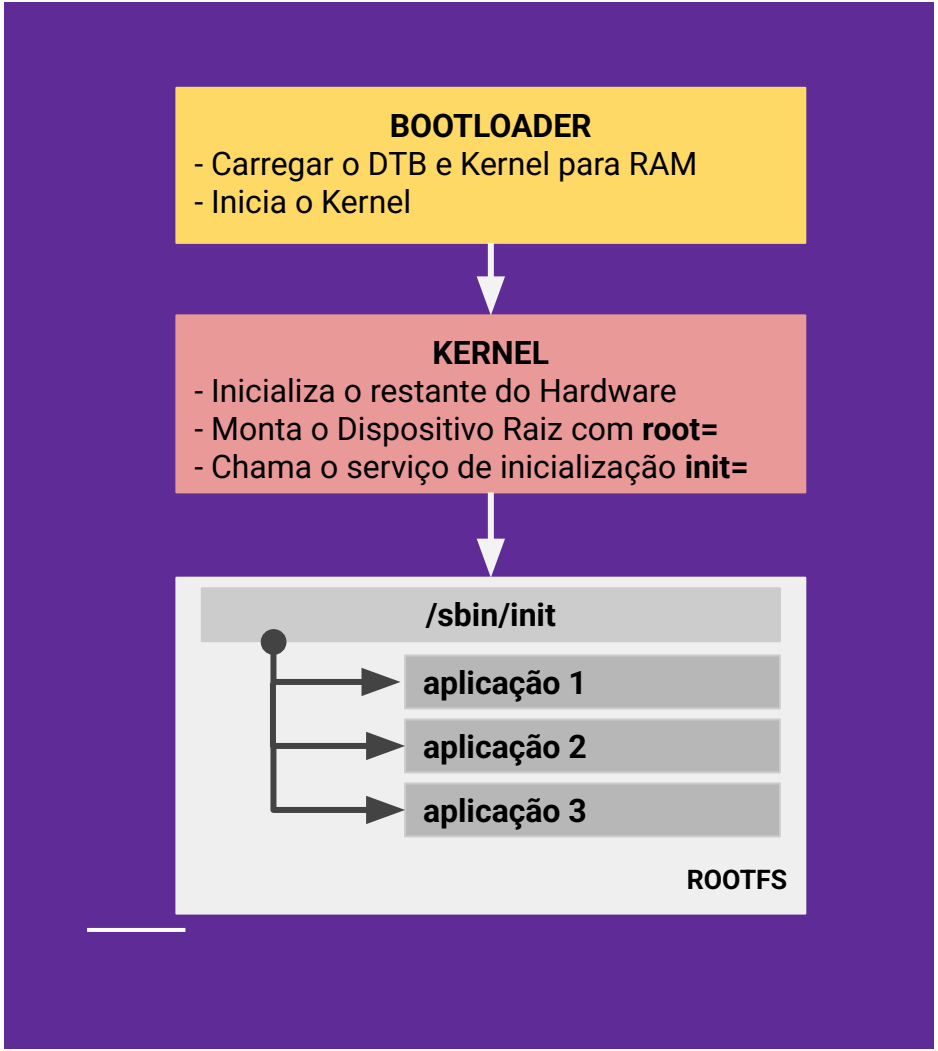
- Comando para montar **/sys**: **mount -t sysfs nodev /sys**

Sistema de Arquivo Virtual - SysFS

- Alguns exemplos e informações úteis do **sysfs**:

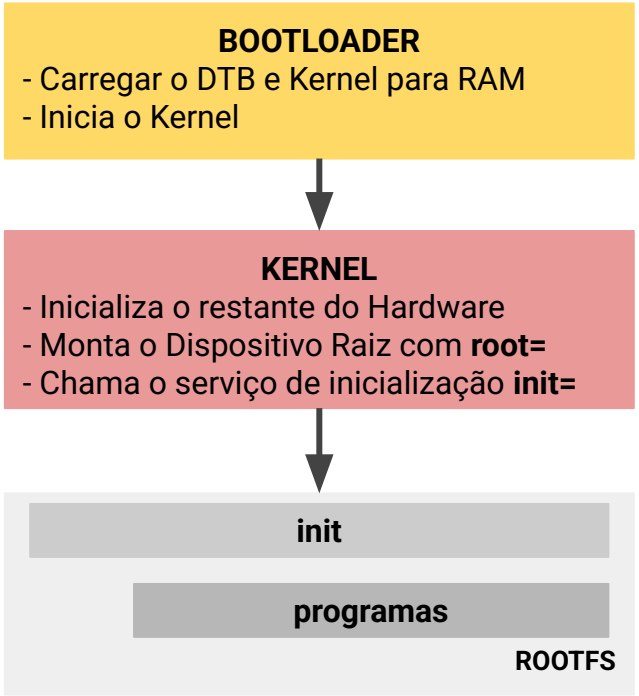
/sys/block	Contém subdiretórios para cada sistema de arquivos de bloco (discos rígidos, etc) com diversas informações.
/sys/bus	Contém subdiretórios para cada tipo de barramento suportado pelo Kernel e se expande em diretórios de devices e drivers.
/sys/class	Contém representações de cada classe de dispositivo que está registrado no Kernel: **pwm, gpio, net, i2c, etc.**
/sys/kernel	Contém vários arquivos e subdiretórios que fornecem informações sobre o kernel em execução.
/sys/module	Contém subdiretórios com todos os módulos carregador do Kernel em memória, permite configurações e obter detalhes.
/sys/fs	Contém subdiretórios para alguns sistemas de arquivos do SO, permite obter diversas informações.

Inicialização Básica no Linux



Inicialização Linux - init

- Assim que o Kernel monta o dispositivo especificado em **root=**, é executado o processo de inicialização **init**.
- O init é pesquisado na seguinte ordem:
  - /sbin/init
  - /bin/init
  - /etc/init
  - /bin/sh
- Pode ser especificado manualmente para o kernel com: **init=/bin/startup.sh**

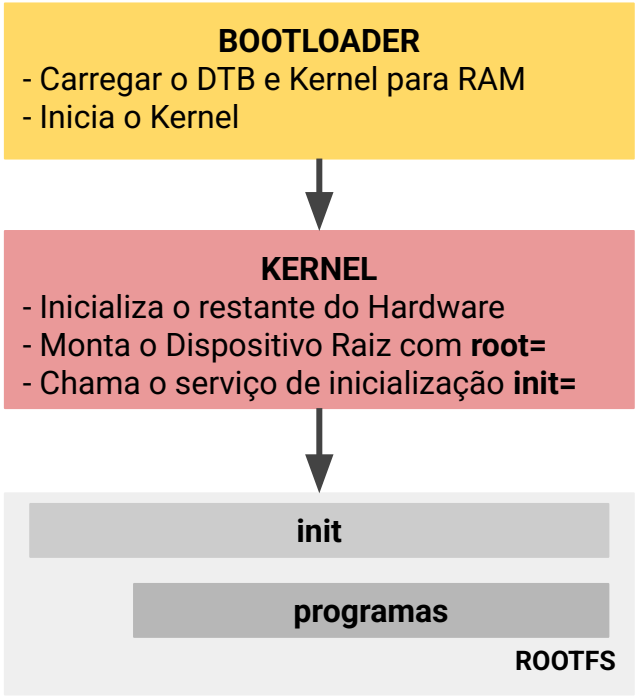


Inicialização Linux - init

- Caso nenhuma das alternativas de init seja encontrada a seguinte mensagem de pânico é lançada:

> **Kernel panic** - not syncing: No init found. Try passing init= opt on to kernel

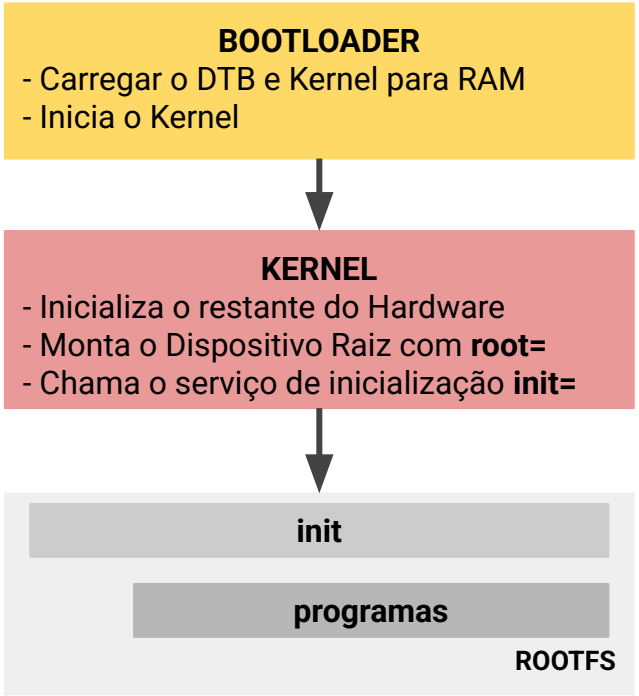
- Se executado com sucesso, é dado início a inicialização de diversos serviços em sequência ou paralelo (Rede, Aplicações, Servidor Gráfico, ...), responsabilidade está dos **Mecanismos de Inicialização**



Inicialização Linux - Mecanismo de Inicialização

- **systemd** - Desenvolvido por Engenheiros da RedHat em meados de 2009, logo adotado pelo Ubuntu e demais Distribuições, vale destacar inicialização paralela e diversas ferramentas

- **sysvinit** - O mais simples e prático, puramente baseado em Script Shell, carrega as configurações em **/etc/inittab** e inicializa os script em **/etc/init.d**



Inicialização Linux - sysvinit

- /etc/inittab

```
$ cat /etc/inittab
# When the system starts
::sysinit:/etc/init.d/rcS

# When pressing Ctrl+Alt+Del
::ctrlaltdel:/sbin/reboot

::respawn:-/bin/sh

# When the system is shut down
::shutdown:/bin/umount -a -r

# When the system restarts
::restart:/sbin/init
```

Inicialização Linux - sysvinit

- /etc/init.d/rcS

```
$ cat /etc/init.d/rcS
#! /bin/sh

/bin/mount -a

/bin/mkdir -p /dev/pts
/bin/mount -t devpts devpts /dev/pts
/bin/mount -t sysfs  sysfs  /sys

/bin/echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s

/bin/mount -o remount, rw /
```

Inicialização Linux - sysvinit

- /etc/init.d/rcS

```
root@b2 # cat /etc/init.d/rcS
...
for i in /etc/rc$runlevel.d/S*
do
        case "$i" in
                *.sh)
                        . $scriptname
                        ;;
                *)
                        ”$@"
                        ;;
        esac

done
```

Inicialização Linux - SystemD

- /etc/systemd/system/aplicacaoGPS.service

```
$ cat /etc/systemd/system/aplicacaoGPS.service
[Unit]
Description=Aplicacao Demo GPS
After=multi-user.target

[Service]
Type=simple
Environment=DEBUG=0
ExecStart=/usr/bin/aplicacaoGPS

[Install]
WantedBy=multi-user.target
```

Inicialização Linux - SystemD

- Iniciando o serviço **aplicacaoGPS.service**

```
$ systemctl start aplicacaoGPS
```

- Parando o serviço **aplicacaoGPS.service**

```
$ systemctl stop aplicacaoGPS
```

- Status do serviço **aplicacaoGPS.service**

```
$ systemctl status aplicacaoGPS
```

Inicialização Linux - SystemD

- Desabilitando o serviço **aplicacaoGPS.service**

```
$ systemctl disable aplicacaoGPS
```

- Habilitando o serviço **aplicacaoGPS.service**

```
$ systemctl enable aplicacaoGPS
```

- Verifica toda configuração do serviço **aplicacaoGPS.service**

```
$ systemctl show aplicacaoGPS
```

Inicialização Linux - SystemD

- Obtendo a configuração e localização do serviço:

```
$ systemctl cat aplicacaoGPS
```

BusyBox

O básico para rootfs em um único binário!

Por que BusyBox?

- Um sistema Linux precisa de poucos programas para funcionar:
  - O programa init
  - Um Interpretador Shell
  - Alguns utilitários básicos para manipulação de arquivos e configuração do sistema(ps, top, cat, echo, grep, ...)
- Em sistemas GNU/Linux normais, esses programas são fornecidos por projetos diferentes: coreutils, bash, grep, etc.
  - Muitos componentes diferentes para integrar
  - Componentes não projetados com restrições de sistemas embarcados em mente

Por que BusyBox?

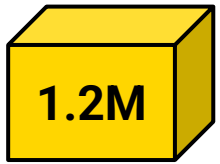
- **BusyBox** é uma solução alternativa, extremamente comum em sistemas embarcados, e facilita **todos projetos em um único binário**!
- Completamente configurável os projetos que devem compor
- Licença: GNU GPLv2(mesma do Kernel Linux)
- Conhecido como ***Swiss Army Knife of Embedded Linux***
- Busybox: https://www.busybox.net
- Alternativa: **Toybox**(Android >6) - Licença BSD

Por que BusyBox?

- Todos os utilitários são compilados em um único executável, **/bin/busybox**
  - Links simbólicos para **/bin/busybox** são criados para cada aplicativo integrado ao BusyBox

- Uma excelente ferramenta para Sistemas Embarcados:

  /bin/busybox (compilado com linkagem estática)



Por que BusyBox?

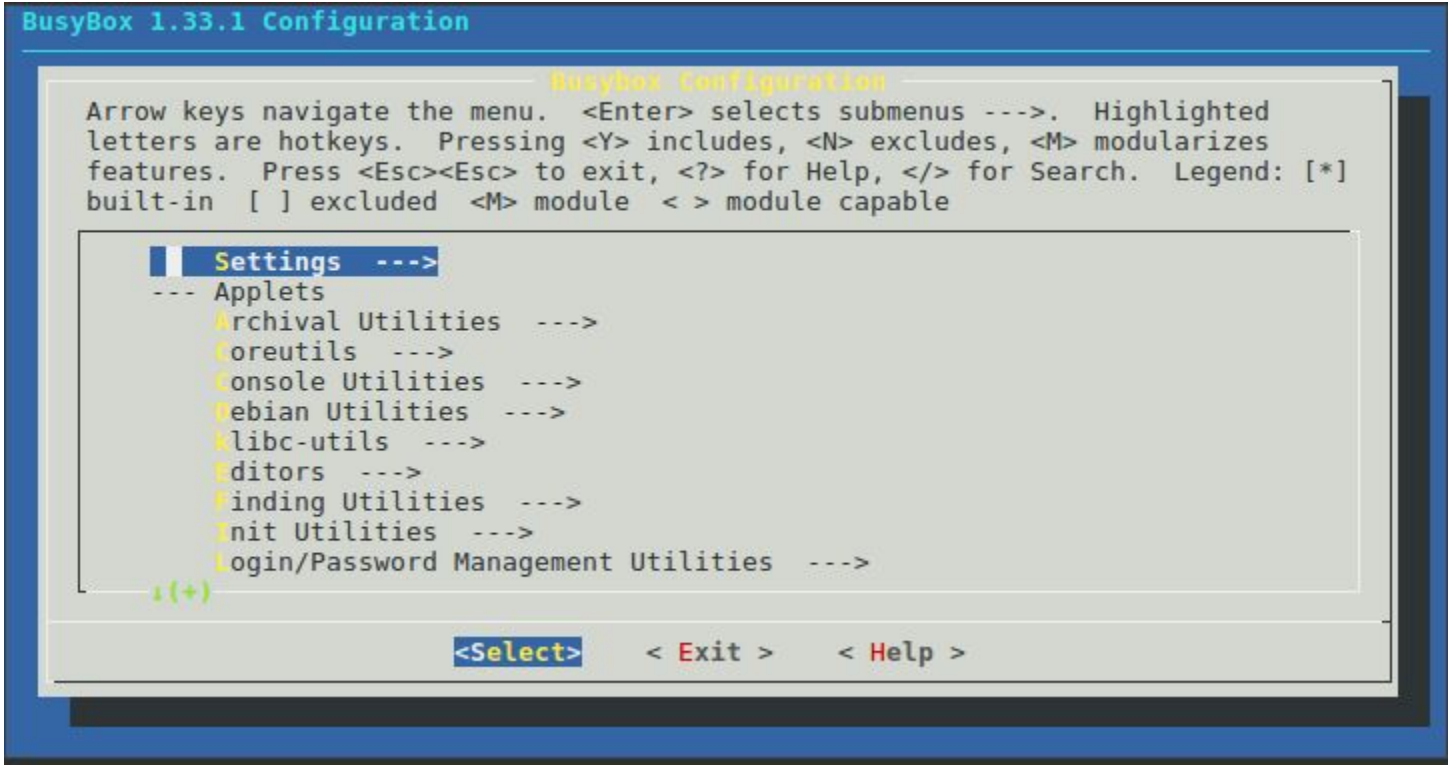
```
_install/
├── bin
│   ├── ash -> busybox
│   ├── busybox
│   ├── cat -> busybox
│   ├── date -> busybox
│   └── ps -> busybox
├── sbin
│   ├── init -> ../bin/busybox
│   ├── poweroff -> ../bin/busybox
│   └── reboot -> ../bin/busybox
└── usr
    ├── bin
    │   ├── top -> ../../bin/busybox
    │   ├── uptime -> ../../bin/busybox
    │   └── wget -> ../../bin/busybox
    └── sbin
        └── udhcpd -> ../../bin/busybox
```

Por que BusyBox?

```
root@tdxBoard # busybox
BusyBox v1.36.1 (2024-01-13 15:57:21 -03) multi-call binary.

Currently defined functions:
        [, [[, adduser, arp, ash, awk, base64, basename, bc, blkid, blockdev, cat, chattr, chgrp, chmod,
chown, chroot, chvt, clear, cp, crond, crontab, cut, date, dd, deluser, depmod, devmem, df, dmesg, du,
echo, env, envdir, false, fdisk, find, free, fsck, fsck.minix, getty, grep, groups, gzip, halt, head,
hexdump, hostname, httpd, hwclock, id, ifconfig, ifdown, ifup, inetd, init, insmod, kill, killall, klogd,
last, linux64, linuxrc, ln, logger, login, ls, lsattr, lsmod, lsof, lsusb, makedevs, man, md5sum, mdev,
mesg, mkdir, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.vfat, mknod, modinfo, modprobe, more,
mount, mv, nc, netstat, nice, nohup, nologin, nproc, ntpd, od, openvt, passwd, patch, pidof, ping, pkill,
pmap, poweroff, printf, ps, pscan, pstree, pwd, reboot, renice, reset, rm, route, run-init, runlevel, sed,
seq, setserial, sh, sha256sum, sleep, sort, start-stop-daemon, stat, strings, stty, su, sulogin, swapoff,
swapon, switch_root, sync, sysctl, syslogd, tail, tar, taskset, tcpsvd, telnet, telnetd, tftp, tftpd, time,
top, touch, traceroute, tree, true, tty, ttysize, udhcpc, udhcpd, uevent, umount, uname, unzip, uptime,
users, vi, w, wc, wget, who, whoami, xz, yes, zcat
```

Por que BusyBox?

- Configurando: **make menuconfig**



Por que BusyBox?

- Compilando:

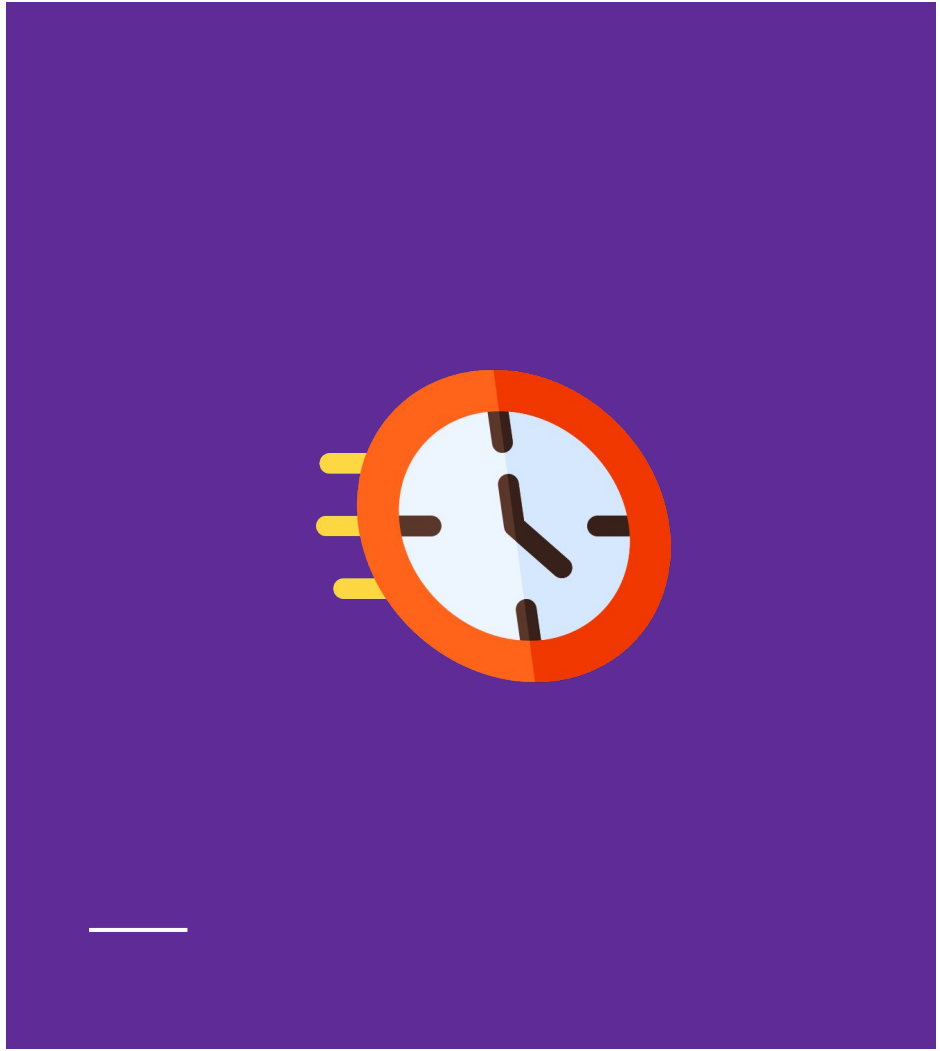
```
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

- Instalando:

```
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi- install
```

  - Um diretório chamado _install será criado e utilizado para instalar o binário e os links-simbólicos.

BusyBox Init



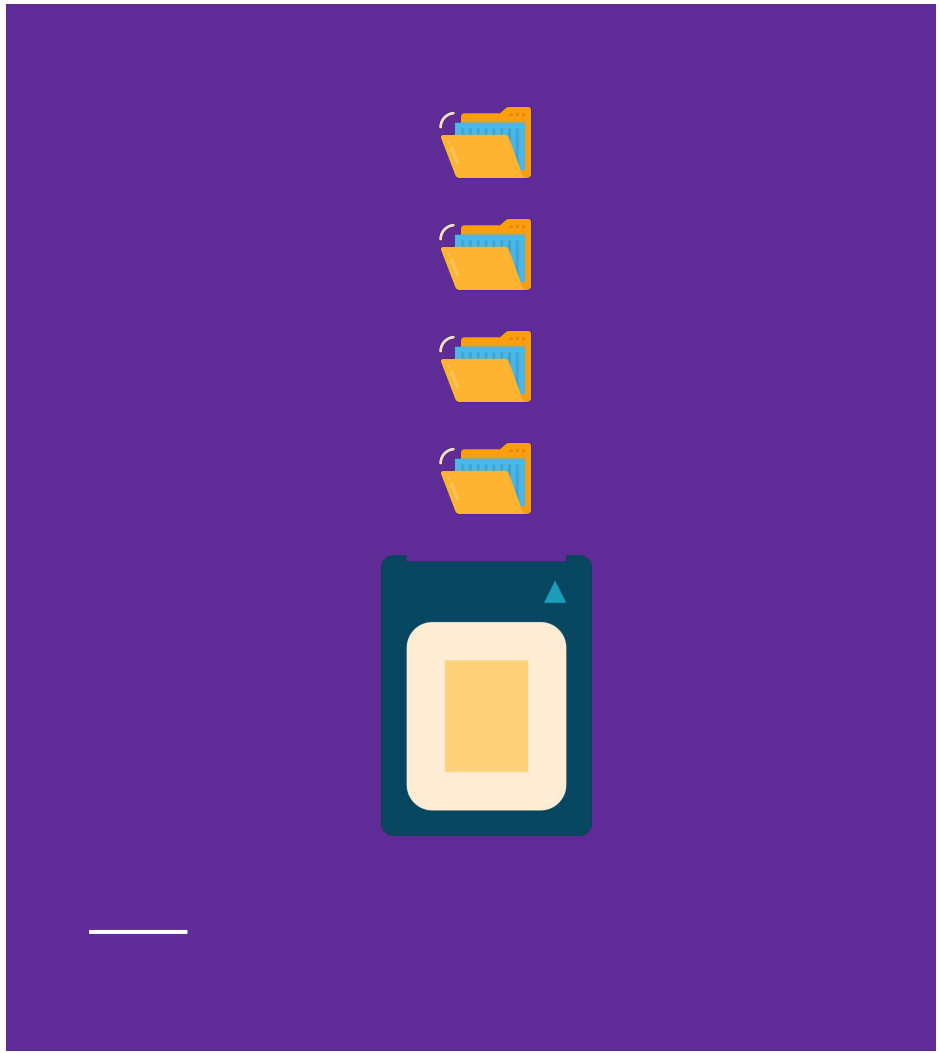
BusyBox - init

- BusyBox fornece uma implementação de um programa init
- Mais simples do que a implementação **init** encontrada em sistemas como (SysVinit ou SystemD)
- Um único arquivo de configuração: **/etc/inittab**
  - Cada linha tem o formato **<id> :: <action>: <process>**
  - BusyBox init não suporta runlevels, se você precisa de runlevels, use sysvinit
- Permite executar serviços na inicialização e garantir que determinados serviços estejam sempre em execução no sistema
- Exemplo de inittab - Bootlin inittab

Laboratório

- Baixando, Configurando e compilando o BusyBox para ARM e criando RootFS
- Carregando o Kernel e RootFS no Bootloader
- Primeiro Boot completo via Rede com TFTP e NFS

Unidade de Armazenamento e Sistemas de Arquivos



Unidades de Armazenamento

- Os dispositivos de armazenamento são classificados em dois tipos principais: **Dispositivos de Bloco** e **Dispositivos Flash**
  - Eles são tratados por diferentes subsistemas e diferentes sistemas de arquivos
- **Dispositivos de Bloco** - Podem ser lidos e gravados em uma base por bloco, em ordem aleatória, sem apagar.
  - Discos rígidos
  - Pendrive, SSD, cartões SD, eMMC: Baseiam-se no armazenamento Flash, mas possuem um controlador integrado que emula um dispositivo de bloco..
- **Dispositivos Flash RAW** - São gerenciados por um controlador no SoC. Eles podem ser lidos, mas a escrita requer limpar(apagar).
  - Flash NOR, flash NAND

Listagem de Dispositivos e Partições

- A lista de todos os dispositivos de bloco disponíveis no sistema pode ser encontrada em **/proc/partitions**

```
$ cat /proc/partitions
major  minor  #blocks  name
   8        0  976762584 sda
   8        1   48827392 sda1
   8        2  244141056 sda2
   8        3  195312640 sda3
   8        4  488479744 sda4
 179        0    7761920 mmcblk0
 179        1      85196 mmcblk0p1
 179        2    3670016 mmcblk0p2
```

- **/sys/block/** também armazena informações sobre cada dispositivo de bloco, por exemplo, se é um armazenamento removível ou não.

Particionamento

- Dispositivos de bloco podem ser particionados para armazenar diferentes partes de um sistema, exemplo:
  - /dev/mmcblk0 - O dispositivo
  - **/dev/mmcblk0p1**- Primeira Partição do Dispositivo - **Bootloader e Kernel**
  - **/dev/mmcblk0p2**- Segunda Partição do Dispositivo - **RootFS**
  - **/dev/mmcblk0p3**- Terceira Partição do Dispositivo - **Banco de Dados/Configurações**

- Algumas ferramentas para criar e modificar as partições em um dispositivo de bloco: fdisk, cfdisk, sfdisk, parted, gparted, etc.

Transferindo Dados

- Para transferir dados para um Dispositivo de Bloco requer um procedimento antes chamado de montar o dispositivo, comando **mount**
- Com o comando mount o Dispositivo(**/dev/mmcblk0p2**) passa ser acessando como, por exemplo (**/mnt/rootfs**)

```
# mkdir /mnt/rootfs

# mount -t ext4    /dev/mmcblk0p2    /mnt/rootfs
```

- Após o ponto de montagem pode ser utilizada ferramentas como cp e mv para transferir arquivos para o Dispositivo
- Após seu uso, utilizar **umount** para desmontar o Dispositivo

Transferindo aDdos

- Ao executar somente o comando mount, irá exibir todos pontos de montagem do sistema

```
$ mount
...
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=4016184k,mode=755)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=807844k,mode=755)
tmpfs on /run type tmpfs (rw,nosuid,relatime,size=262144k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/mmcblk0p2 on /mnt/rootfs type ext4 (rw,relatime, data=ordered)
/dev/mmcblk0p1 on /mnt/boot   type vfat (rw,relatime,errors=remount-ro)
...
```

Transferindo Dados

- Muitas vezes é necessário transferir dados de ou para um dispositivo de bloco de forma RAW
  - Especialmente para gravar uma imagem do sistema de arquivos(com partições) em um dispositivo de bloco
- Os dispositivos de bloco em **/dev/** permitem tal acesso bruto
- Uma das principais e mais conhecida ferramenta é o **dd** (***d**isk **d**uplicate*):

```
# dd if=/dev/mmcblk0p1 of=test.raw bs=1M count=32
```

  - Copia os primeiros 32 blocos de 1 MB de /dev/mmcblk0p1 para o arquivo de test.raw

```
# dd if=test.raw of=/dev/sda2 bs=1M seek=4
```

  - Copia o conteúdo completo do test.raw para /dev/sda2, por blocos de 1 MB, mas começando com deslocamento de 4MB em /dev/sda2

Transferindo Dados

- Exemplo com um arquivo de imagem:

```
$ fdisk -l raspios-buster-armhf-lite.img
Disk raspios-buster-armhf-lite.img: 1,8 GiB, 1874853888 bytes,Sector size
...

Dispositivo                    Inicializar  Start    Fim Setores  Size  ID  Tipo
raspios-buster-armhf-lite.img1       8192   532479      524288  256M  c   W95 FAT32 (LBA)
raspios-buster-armhf-lite.img2     532480  3661823     3129344  1,5G  83  Linux


$ sudo dd if=raspios-buster-armhf-lite.img of=/dev/mmcblk0 bs=1M
```

- bmap-tools é uma alternativa ao dd mais eficiente e com mais recursos.

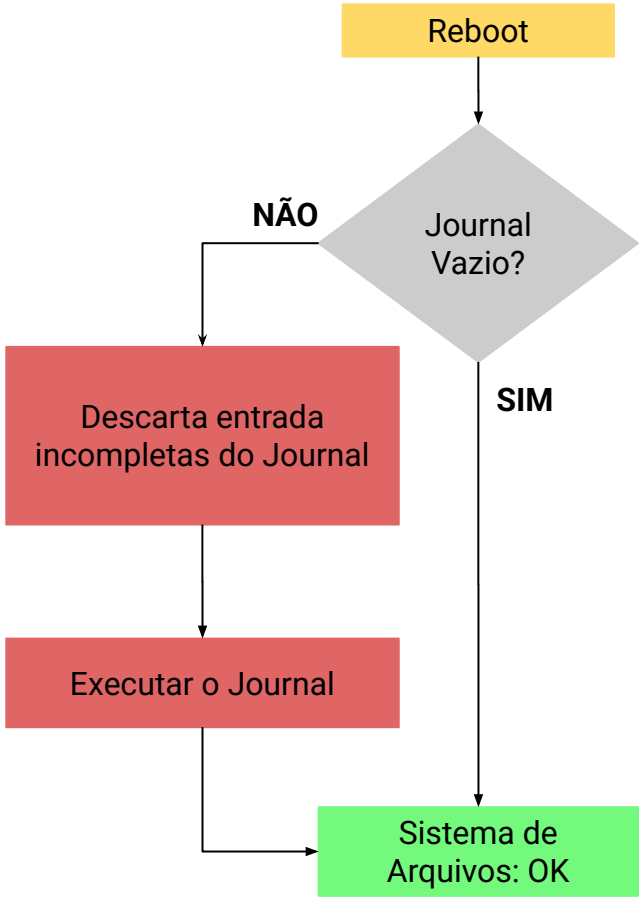
Sistemas de Arquivos

- O sistema de arquivos padrão usado em Sistemas Linux é:
  - **ext2** (utilizado por mais de 10 anos, ainda em uso em Sistemas Embarcados)
  - **ext3** (trouxe o journaling em comparação com o ext2, agora obsoleto com o ext4)
  - **ext4**(melhorias de desempenho e suporte para partições muito grandes)

- Suporta todos os recursos que o Linux precisa em um sistema de arquivos raiz: permissões, propriedade, arquivos de dispositivo, links simbólicos, etc.

Sistemas de Arquivos - Journaling

- Alguns Sistemas de Arquivos com suporte a journal:
  - ext3/ext4
  - btrfs
  - XFS
  - ReiserFS
  - ZFS



Sistemas de Arquivos com Compressão

- Há casos que um Sistema de Arquivos com compressão seja requisitado, por questão de espaço por exemplo:

  - SquashFS
  - EROFS
  - tmpFS

Sistemas de Arquivos com Compressão - SquashFS

- Sistema de Arquivos compactado somente leitura(read-only) para dispositivos de bloco. Bom para partes de um sistema de arquivos que podem ser somente leitura (kernel, binários ...)
- Ótima taxa de compressão, o que geralmente traz um melhor desempenho de leitura
- Suporta vários algoritmos de compressão (LZO, XZ, etc.)
- Benchmarks:
  - https://elinux.org/Squash_Fs_Comparisons
- Mais informações em SquashFS 4.0 FileSystem

Sistemas de Arquivos com Compressão - EROFS

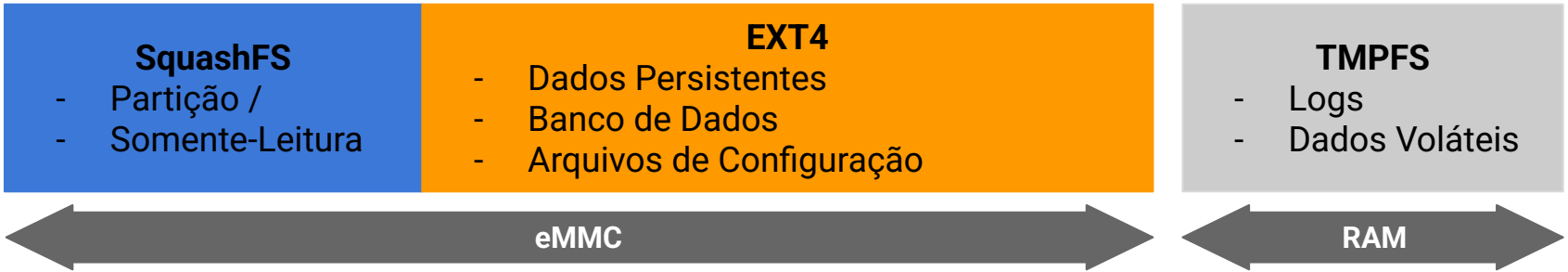
- Uma futura alternativa ao SquashFS
- Desenvolvido pela Huawei - **EROFS** (Enhanced Read-Only File System)
- Huawei introduziu em todos os seus produtos com Android**[1]**
- Introduzido no Kernel Linux 5.4
- Algoritmo de Compressão **LZ4**
- Tamanho Máximo de Arquivo Comprimido 4GB
- Mais informações em Enhanced Read-Only File System - EROFS

Sistemas de Arquivos com Compressão - TMPFS

- Não é um sistema de arquivos em bloco, é claro!
- Perfeito para armazenar dados temporários na RAM: arquivos de log do sistema, dados de conexão, arquivos temporários ...
- Mais eficiente em termos de espaço do que discos, os arquivos estão diretamente no cache de arquivos, aumenta e diminui dinamicamente
- Exemplos:
  - mount -t tmpfs run /var/run
  - mount -t tmpfs shm /dev/shm
- Mais informações em TmpFS

Sistemas de Arquivos - Exemplo em Projeto

- É aconselhável e boa prática e mistura de Sistemas de Arquivos



Memória Flash

- Flash NAND
  - SLC
  - MLC
- MTD
- UBI/UBIFS

Memória Flash NAND

- Permite acesso aleatório aos dados
- As células Flash NAND são categorizadas dependendo do número de bits que podem armazenar:
  - **SLC(Single Level Cell)** - Pode armazenar um bit por célula, mais rápido, tempo de vida maior, o mais caro de todos, cerca de 100 mil ciclos.
  - **MLC(Multi Layer Cell)** - Pode armazenar dois bits por célula, mais estável que TLC, menos durável e estável como que SLC, de 3 mil a 15 mil ciclos.
  - **TLC(Triple Layer Cell)** - Pode armazenar três bits por célula, custo de produção mais barato, processo de escrita mais lento, cerca de 500 a 1000 ciclos de gravação.

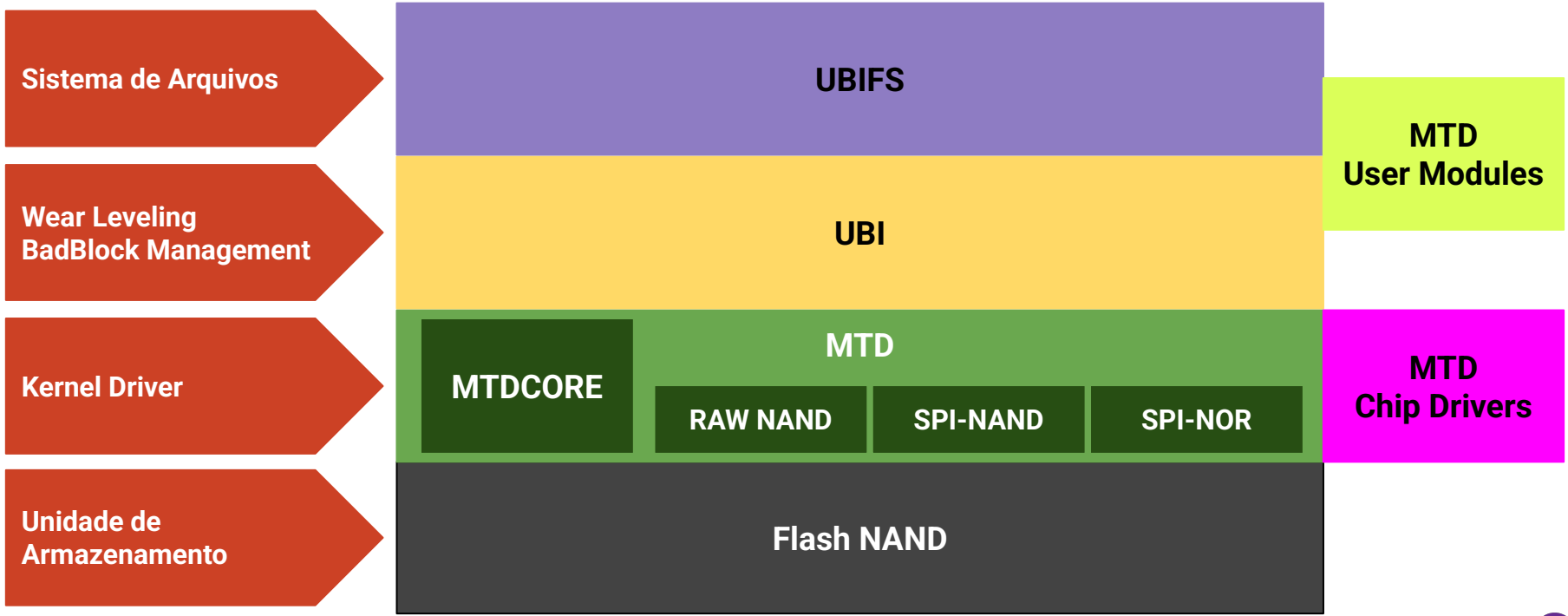
Memória Flash NAND

- Vida útil curta em comparação com outras formas de armazenamento
- A vida útil depende da tecnología Flash NAND (SLC, MLC): entre 1.000.000 e 1.000 ciclos de apagamento por bloco
- **Wear Leveling**(Mecanismos de nivelamento de desgaste) são necessários para apagar blocos uniformemente
- **Bad Block Management**(Detecção/Gerenciamento de blocos defeituosos)
- 1 Ciclo para leitura e 2 Ciclos para escrita(apagar e escrever)

Memória Flash NAND - Subsistema MTD

- MTD significa Dispositivos de Tecnologia de Memória
- Subsistema genérico no Linux lidando com todos os tipos de dispositivos de armazenamento que não se sejam de Subsistema de Bloco
- Tipos de dispositivos suportados: RAM, ROM, Flash NOR, Flash NAND
- Características da mídia de armazenamento abstrata e fornece uma API simples para acessar dispositivos MTD

Memória Flash NAND - Subsistema MTD



Memória Flash NAND - MTD User Modules

- O driver **mtdchar** cria um dispositivo de caractere para cada device/partição MTD do sistema
  - /dev/mtdX
  - /dev/mtdXro
  - Permite comando ioctl() para apagar e manipular a Flash
- O driver **mtdblock** implementa o módulo "block device" da flash.
  - Normalmente /dev/mtdblockXY
  - Permite acesso de leitura/gravação igual em disco, no entanto, não faz BadBlock Management e Wear Leveling
  - Para isso utiliza um Sistema de Arquivos que faça isso: JFFS2, YAFFS2 e UBIFS

Memória Flash NAND - JFFS2

- Suporta compressão em tempo real
- O Sistema de Arquivos Flash NAND mais antigo e ainda em uso
- Suporte Wear Leveling, ECC, o tempo de inicialização depende do tamanho do sistema de arquivos: não escalona para partições grandes porque precisa verificar todo o armazenamento no momento da inicialização. É necessário ativar **CONFIG_JFFS2_SUMMARY** para resolver este problema.
- Mais informações em JFFS2

Memória Flash NAND - YAFFS2

- Suporta principalmente flash NAND
- Sem compressão
- Suporte a Wear Leveling e resistente a falha de energia
- Tempo de inicialização rápido
- Não faz parte do kernel oficial do Linux
- Mais informações em YAFFS2

Memória Flash NAND - UBIFS

- UBI - Unsorted Block Images
- Destina-se a substituir **JFFS2**, com diversas melhorias
- Camadas de Wear Leveling e Sistemas de Arquivos separadas, Wear Leveling opera em todo armazenamento e não por partição
- Foco em escalabilidade, desempenho e confiabilidade
- Desvantagem: Introduz uma sobrecarga de espaço perceptível, especialmente quando usado em pequenos dispositivos ou partições, o JFFS2 ainda faz sentido em pequenas partições MTD.
- Os volumes podem ser redimensionados como dinâmico ou estático(ro)
- Mais informações em UBI

Memória Flash NAND - Dicas de uso

- Ao se trabalhar com memórias Flash também se deve levar em consideração a vida útil limitada dos dispositivos flash, tomando precauções adicionais:

  - Não use seu armazenamento flash como swap(**swapfile**)
  - Monte seus sistemas de arquivos como somente leitura(read-only) sempre que possível, ou utiliza um Sistema de Arquivos como **SquashFS**
  - Mantenha os arquivos voláteis na RAM (tmpfs)
  - Não use a opção **sync** no ponto de montagem (confirma as gravações imediatamente). Use a chamada de sistema **fsync()** para sincronização por arquivo

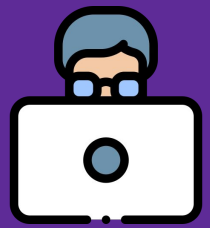
Laboratório

- Criando microSD da Instalação com TEZI
- Realizando boot!

Laboratório

Prática 02

- Configurando um DHCP Server ou DHCP Cliente com Busybox



Laboratório

Prática 03

- Instalando e Configurando Acesso via Rede
- Transferindo e Executando os binários da **Prática 01**

Laboratório

Prática 04 - Device-Tree

- Analisando a estrutura via Linux
- Analisando Periféricos
- Modificando Periféricos
- Compilando Novo Device-Tree

Desenvolvendo Aplicações em Linux Embarcado

Componentes Open-Source

- Linguagens Interpretadas
- WebServers
- Banco de Dados
- Bibliotecas Gráficas
- Bibliotecas Multimédia
- Utilitários para o Sistema

Escolhendo componentes Open-Source

- Procure por projetos ativo e com diversos colaboradores.
- Se atente sobre a LICENÇA do projeto - **Atenção com *GPLv3.**
- Analise os releases do projeto e periodicidade de atualizações.
- Estude sobre a qualidade do projeto, plataformas suportadas, sistemas de construção, se possui Testes Unitários, etc.

Linguagens Interpretadas

- Suporte das principais linguagens interpretadas:
  - Shell Script
  - Python
  - Perl
  - Ruby
  - PHP
  - Lua
- Além de Linguagens Compiladas e Modernas como:
  - C/C++
  - Java
  - Go
  - Rust

WebServers

- Alguns dos principais WebServers:
  - Busybox http
  - Apache2
  - Lighttpd
  - Nginx (lowjs versão otimizada para Sistemas Embarcados)
  - Monkey
  - Cherokee
  - Nodejs
  - Python3 http.server
  - Python Flask

Banco de Dados

- Alguns dos principais Banco de Dados:
  - SQLite - O mais popular e indicado em Sistemas Embarcados
  - PostgreSQL
  - hsqldb
  - mariadb

Bibliotecas Gráficas

- Para a maioria das aplicações gráficas funcionarem é necessário um “Servidor Gráfico”, digamos que seriam Bibliotecas Gráficas mais baixo nível:

  - Xorg(Kdrive)

  - Wayland

Bibliotecas Gráficas

- Para a maioria das aplicações gráficas funcionarem é necessário um “Servidor Gráfico”, digamos que seriam Bibliotecas Gráficas mais baixo nível:

  - Xorg(Kdrive)
    - Versão simplificada do servidor X, para sistemas embarcados
    - Funciona em cima do Linux framebuffer, graças à variante Xfbdev
    - Permite usar qualquer aplicativo ou biblioteca X11 existente
    - Licença X11

Bibliotecas Gráficas - Wayland

- Para a maioria das aplicações gráficas funcionarem é necessário um “Servidor Gráfico”, digamos que seriam Bibliotecas Gráficas mais baixo nível:

  - Wayland
    - Uma substituição mais simples para o X
    - Wayland é um protocolo para um compositor falar com seus clientes.
    - Weston: uma implementação de referência mínima e rápida de um compositor Wayland, e é adequado para muitos casos de Sistemas Embarcados

Bibliotecas Gráficas - Framework Qt

- Um dos mais famosos frameworks gráficos para Sistemas Embarcados
- Multiplataforma: Windows, Linux, Android, MacOS, entre outros
- Implementado em C++
- Binding Oficial de Python e de outras linguagens mantido pela comunidade
- **Qt Widgets**(Desktop Style) e **Qt QML** (QtQuick - Aceleração Grafica)
- QtCreator IDE Oficial
- Qt Platform Abstract
  - **XCB**
  - **Wayland**/Qt Wayland
  - **EGLFS** - Sem Servidor Gráfico e com suporte a aceleração gráfica
  - **LinuxFB** - Sem Servidor Gráfico e sem necessidade de GPU

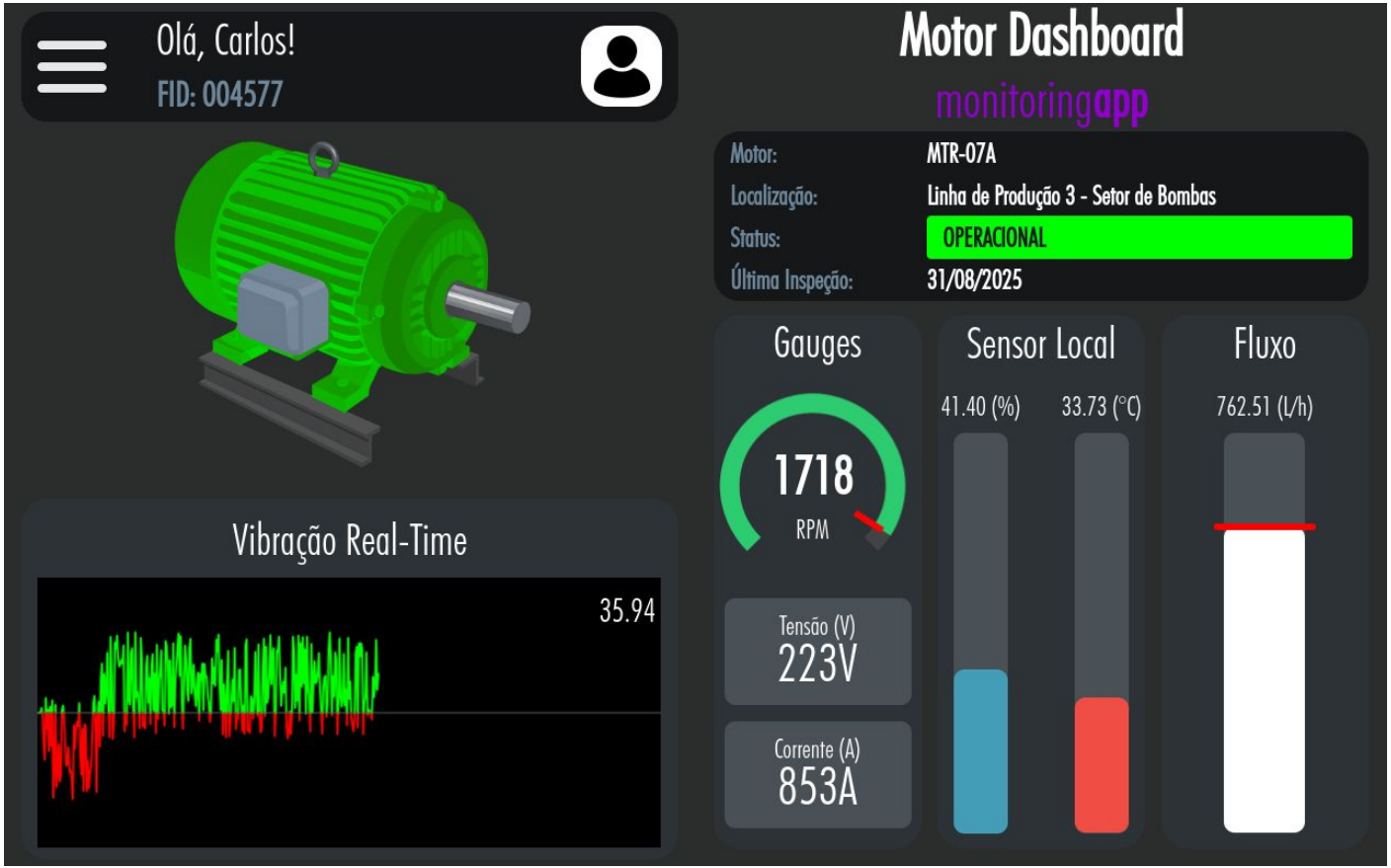
Bibliotecas Gráficas - Framework Qt

- Licença Dual: Comercial e Open-Source

Demos B2Open Systems

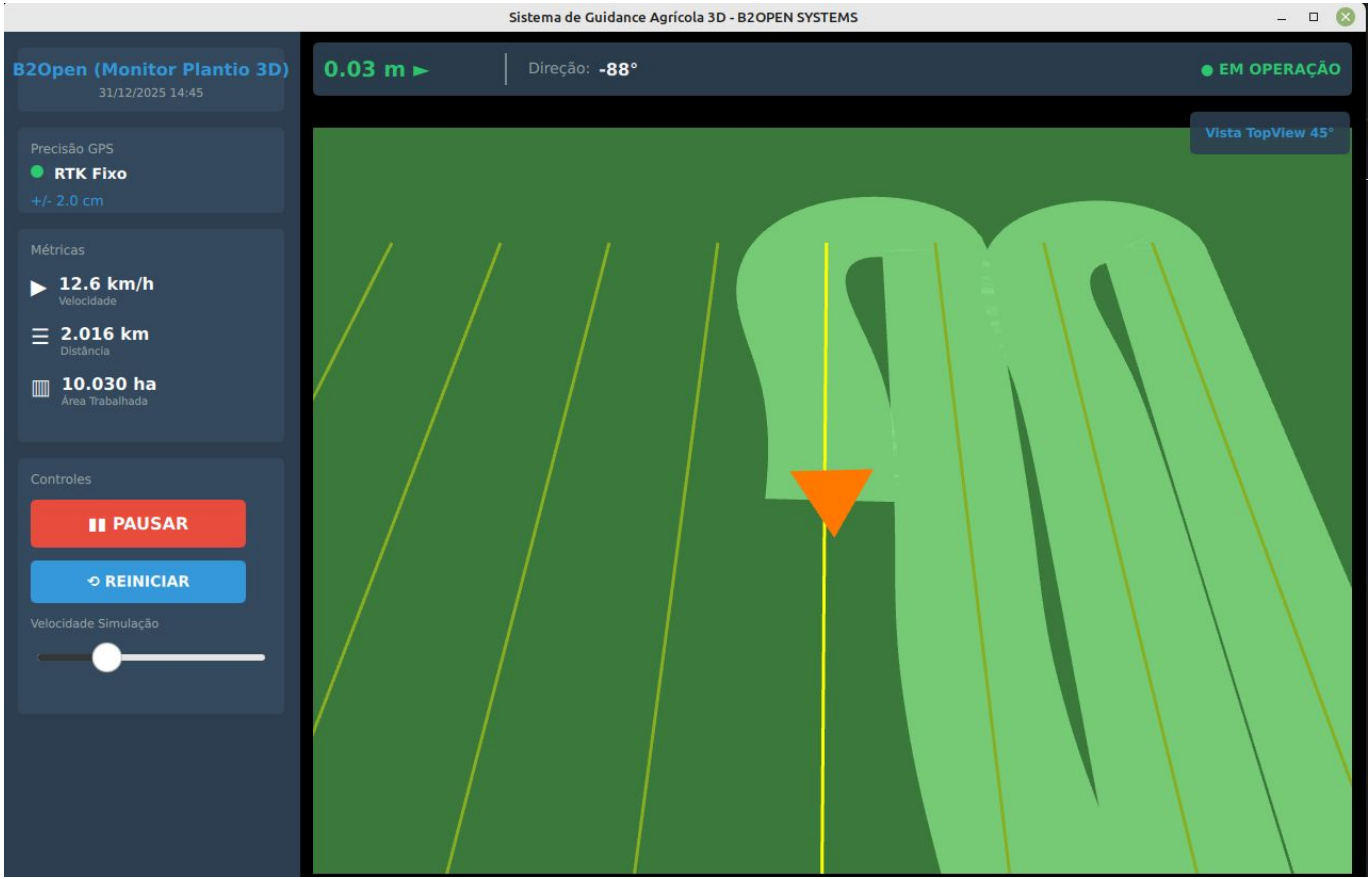
Bibliotecas Gráficas - Framework Qt

Demos B2Open Systems



Bibliotecas Gráficas - Framework Qt

Demos B2Open Systems



Bibliotecas Gráficas - LVGL

- LVGL - Light and Versatile Graphics Library
- Baixo consumo de Memória, surgiu inicialmente para MCU
- Suporte TouchScreen e Mouse
- Plataforma: Xorg e FrameBuffer
- Mais de 30 controles para ser utilizados
- Linguagem C
- Licença MIT
- https://lvgl.io/

Bibliotecas Gráficas - LVGL

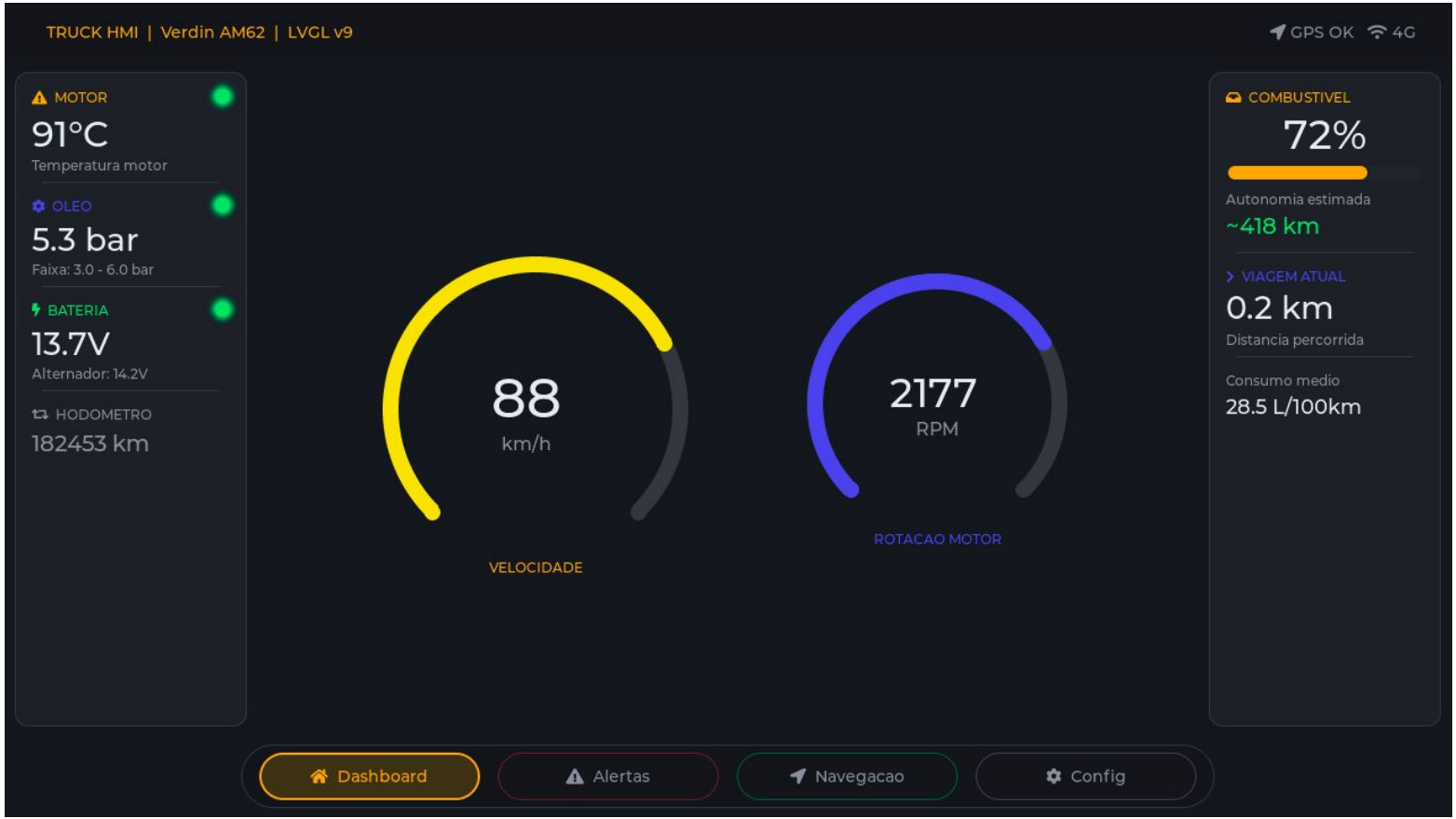
Demos B2Open Systems

Bibliotecas Gráficas - LVGL

Demos B2Open Systems

Bibliotecas Gráficas - LVGL

Demos B2Open Systems



Bibliotecas Gráficas - LVGL

Demos B2Open Systems

Bibliotecas Gráficas - LVGL

Demos B2Open Systems

Bibliotecas Gráficas - FLTK

- FLTK - Fast Light Toolkit
- Multiplataforma Plataforma: Linux(Xorg), Windows e MacOS
- Linguagem C++
- Licença LGPLv2.1
- https://www.fltk.org/

Bibliotecas Gráficas - Slint

- Slint
- Focado em Embarcados e Desktop
- Linguagem C++ e Rust
- Camada de Integração com Yocto Project
- Diversos Widgets
- Diversas opções de Renderes Backend semelhante ao Qt
- Licença Comercial e GPLv3
- https://slint.dev/

Bibliotecas Gráficas - Flutter

- Google Flutter
- É o SDK do Google para criar experiências de usuário bonitas e rápidas para dispositivos móveis, web e desktop
- Suporte para Embarcados
  - https://github.com/sony/flutter-embedded-linux
- Linguagem **Dart**
- Licença BSD-3
- https://flutter.dev/

Bibliotecas Gráficas - Kivy

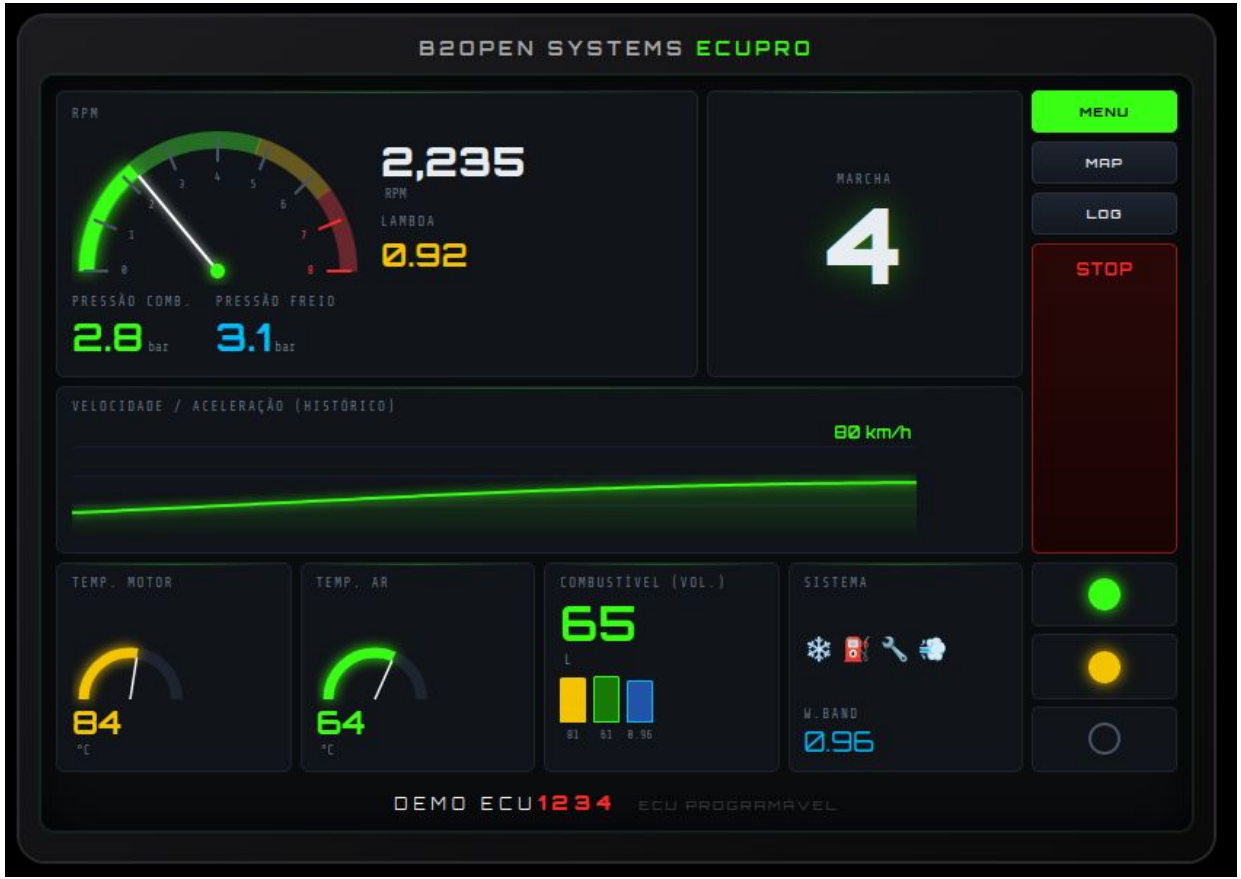
- Framework Kivy
- Suporte X11 e Wayland, além de SDL2 e OpenGL
- Multiplataforma Windows, Linux, Android, MacOS
- Linguagem **Python**
- Licença MIT
- https://kivy.org/
- https://github.com/kivy/kivy

Navegadores

- **WebKit** - Web Browser Engine, Licença LGPL e BSD
- **Qt WebEngine** - Web Browser baseado em Chromium, Licença LGPLv3/GPlv3
- **COG** - Web Browser de baixo consumo e poucos recursos, baseado em WebKit, ideal para Sistemas Embarcados, Licença MIT
- **Surf** - Web Browser baseado em WebKit2 e GTK, Licença MIT

Navegadores

Demos B2Open Systems



Bibliotecas Multimídia

- **Gstreamer** - Um framework para multimídia
  - Suporte encode/decode de diversos codecs
  - Diversas empresas adicional suporte de seus produtos com Gstreamer, ex: Câmeras
- **ALSA** - Principal biblioteca de manipulação de áudio do Linux, utiliza o subsistema ALSA do Kernel
- **Pulseaudio** - Um servidor de áudio com diversos plugins
- **libavcodec** - Projeto de FFMpeg para encode/decode de áudio e vídeo
- **libcamera** - Framework para simplificar acesso e controles de câmeras

Utilitários para o Sistema

❖ Rede

- ➢ openSSH - Acesso Remoto Seguro via Shell)
- ➢ dropbear - SSH mais leve
- ➢ dnsmasq - DNS e DHCP
- ➢ networkmanager e connman - Gestão de Interfaces e Conexões
- ➢ iptables - Linux Firmware e Netfilter
- ➢ iw - Ferramenta para gerenciamento de Wireless
- ➢ vsftpd e proftpd - Servidores FTP
- ➢ openvpn e wireguard - Solução para VPN segura
- ➢ net-tools - Ferramentas como arp, ifconfig, route, hostname, netstat, etc

Utilitários para o Sistema

❖ Sistema

- ➢ gpsd - Serviço para processar dados de GPS e entregar em JSON
- ➢ dbus - IPC utilizado pelo Linux e diversas aplicações como systemd
- ➢ libusb - Biblioteca completa para acessar diversas informações de dispositivos USB via userspace
- ➢ libsoc - Biblioteca que abstrai o acesso e uso de GPIO, I2C, SPI e PWM em C e Python*
- ➢ i2c-tools - Manipulação de I2C via userspace
- ➢ can-utils - Ferramentas para configuração e enviar/receber dados da rede CAN(depende de SocketCAN)
- ➢ libgpiod - Novo suporte para manipular GPIO mais eficiente, kernel >= 5.4
- ➢ bluez5 - Pilha completa para manipulação de dispositivos Bluetooth e Protocolos
- ➢ **Busybox** - O canivete suiço de ferramentas para Sistemas Embarcados

Building System

Automação x Compilação Manual

O Problema da "Compilação Manual"

Baixar, configurar e compilar na mão funciona para aprender, mas falha na produção:

- ✗ **Erro Humano:** Esquecer uma **flag** quebra o build.
- ✗ **Inferno de Dependências:** O software A precisa da libB, que precisa de LibC...
- ✗ **Escalabilidade:** Como replicar esse build em 10 máquinas diferentes?

A Evolução: Make, Autotools e CMake

Make

O "motor" básico. Lê o Makefile e decide o que recompilar.

Comando: **make**

Autotools

O padrão GNU clássico. Focado em portabilidade entre Sistemas UNIX.

Comando: **./configure; make**

CMake

O padrão moderno. Não compila, mas gera os arquivos de build para Make ou Ninja de forma inteligente.

Comando: **cmake .. ; make**

Modernidade: Meson, Ninja e Bazel

Meson + Ninja

A dupla mais rápida da atualidade. Meson é legível (Python-like) e Ninja é um motor de execução de baixo nível focado em velocidade que realmente realiza o trabalho pesado.

Bazel (Google)

Focado em monorepos gigantes. Garante que o build seja 100% reprodutível, independente da máquina.

Resumo: No Linux Embarcado, você encontrará majoritariamente **CMake** e **Meson** em projetos modernos.

Build Systems

Automatizando a Construção do Linux Embarcado

O Desafio do "Do Zero"

Por que o Linux From Scratch (LFS) não escala?

A realidade do LFS

O Custo do Trabalho Manual

Construir um sistema peça por peça é excelente para aprender, mas inviável comercialmente:

- ✗ **Manutenção Impossível:** Uma atualização de libC exige recompilar tudo.
- ✗ **Dependências em Cascata:** Resolver conflitos de biblioteca manualmente gasta dias.
- ✗ **Falta de Reprodutibilidade:** Difícil garantir que dois engenheiros gerem a mesma imagem.

A Solução:

Build Systems

Automação Completa do Bootloader ao RootFS

Buildroot x Yocto Project

Critério	Buildroot	Yocto Project
Filosofia	Minimalismo e Simplicidade	Flexibilidade e Escalabilidade
Configuração	Menuconfig (Kconfig)	Camadas e Receitas (.bb e .bbappend)
Tempo de Build	Muito Rápido (Primeiro Build)	Lento (Primeiro build leva horas e é massiva)
Ideal para...	Prototipagem e dispositivos fixos	Linhas de produtos e frotas

Existem outras soluções como **OpenWRT**, **ELBE**, **Armbian** e **PTXdist**, cada uma voltada para diferentes aplicações, níveis de customização e fluxos de desenvolvimento. Estas ferramentas não serão abordadas neste treinamento.

Yocto Project

O Padrão da Indústria para Linux Embarcado

BuildSystem Yocto Project

A anatomia do Yocto Project

Configuração

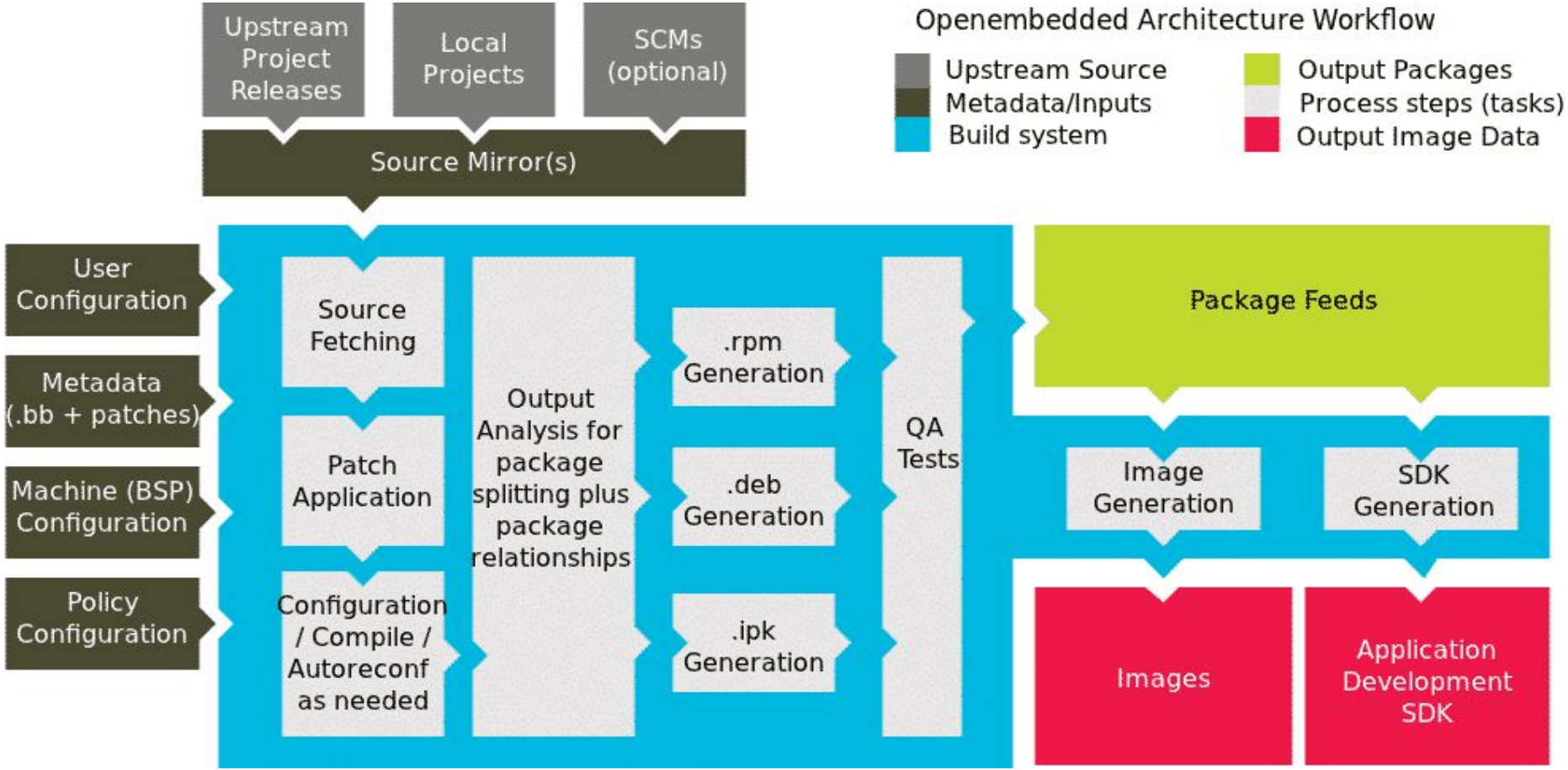
Arquivos .conf definem variáveis globais, o hardware alvo (**MACHINE**) e a política do sistema (**DISTRO**).

Camadas (layers)

Coleções de metadados. Permitem adicionar suporte a hardware (BSP) ou software sem alterar o núcleo do sistema.

Receita (.bb)

Descrevem com obter, compilar e instalar um pacote especifico.


Openembedded Architecture Workflow
Upstream Source
Metadata/Inputs
Build system
Output Packages
Process steps (tasks)
Output Image Data
Upstream Project Releases
Local Projects
SCMs (optional)
Source Mirror(s)
User Configuration
Metadata (.bb + patches)
Machine (BSP) Configuration
Policy Configuration
Source Fetching
Patch Application
Configuration / Compile / Autoreconf as needed
Output Analysis for package splitting plus package relationships
.rpm Generation
.deb Generation
.ipk Generation
QA Tests
Package Feeds
Image Generation
SDK Generation
Images
Application Development SDK

Yocto Project

Obtendo o código-fonte e os primeiros passos

Yocto Project - Releases

CODINOME	YP Version	Release Date	Estado Atual	EOL
Wrynose	6.0	04/2026	LTS	04/2030
Scarthgap	5.0	04/2024	LTS	04/2028
Kirkstone	4.0	05/2022	LTS	04/2026
Dunfell	3.1	04/2020	LTS	04/2024

Dica: Para projetos industriais, prefira sempre versões **LTS** (Long Term Support).

Referência: https://wiki.yoctoproject.org/wiki/Releases

Obtendo o Projeto

```
$ mkdir ~/yp

$ cd ~/yp

~/yp $ git clone -b scarthgap git://git.yoctoproject.org/poky.git
Cloning into 'poky'...
remote: Enumerating objects: 711439, done.
remote: Counting objects: 100% (1804/1804), done.
remote: Compressing objects: 100% (956/956), done.
remote: Total 711439 (delta 1190), reused 1209 (delta 847), pack-reused 709635 (from 1)
Receiving objects: 100% (711439/711439), 222.62 MiB | 7.47 MiB/s, done.
Resolving deltas: 100% (517764/517764), done.
```

Obtendo o Projeto

```
~/yp $ cd poky

~/yp/poky $ git branch
  • scarthgap

~/yp/poky $ ls
bitbake                 MAINTAINERS.md      meta-yocto-bsp          README.qemu.md
contrib                 MEMORIAM            oe-init-build-env       scripts
documentation           meta                README.hardware.md      SECURITY.md
LICENSE                 meta-poky           README.md
LICENSE.GPL-2.0-only    meta-selftest       README.OE-Core.md
LICENSE.MIT             meta-skeleton       README.poky.md
```

Criando um Projeto com Yocto Project

Para iniciar um projeto no Yocto Project e preparar o ambiente, deve-se utilizar o scripts **oe-init-build-env.**

Exemplo:

```
source oe-init-build-env  <nome-diretorio-build >
```

A primeira vez que executar, irá carregar diversas informações e dicas das primeiras images a utilizar, nas demais vezes irá aparecer uma tela resumida.

Primeiro Projeto

```
$ source oe-init-build-env build

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
        core-image-minimal
        core-image-sato
        meta-toolchain
        meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Other commonly useful commands are:
 - 'devtool' and 'recipetool' handle common recipe tasks
 - 'bitbake-layers' handles common layer tasks
 - 'oe-pkgdata-util' handles common target package tasks
```

Primeiro Projeto

Após o source **oe-init-build-env**, o foco está no diretório **conf/**:

- **local.conf** - Configuração local (MACHINE, DISTRO, GERENCIADOR DE PACOTES, LISTA DE PACOTES PARA INSTALAR, CUSTOMIZAÇÕES, ...)

- **bblayers.conf** - Lista das camadas ativas.

- **templateconf.cfg** - Aponta para o template inicial, a mensagem que sempre é exibida após o oe-init-build-env.

Bitbake

Todo trabalho de gerar imagem, toolchain, e processar receitas é realizado pelo **bitbake**.

```
build $ bitbake core-image-minimal
```

Dica: Caso nenhum **MACHINE** seja configurado, uma imagem mínima e funcional será gerada utilizando **QEMU**, e pode testar e executar com um script interno chamado **runqemu**.

Laboratório

Prática 05

- Obtendo o Yocto Project, clonando as camadas, criando um projeto e gerando uma imagem
- Google Repo / Siemens KAS

Laboratório

Prática 06

- Compilar aplicações direto no TARGET - Demo USB

Laboratório

Prática 07

- Manipulando GPIO
- Manipulando PWM
- I2C
- EEPROM

Laboratório

Prática 08

- Servidor Web

Licenças

Licenças Open-Source

- Todo software que está sob uma licença de software livre concede quatro liberdades a todos os usuários
  - Liberdade de uso
  - Liberdade para estudar
  - Liberdade para copiar
  - Liberdade para modificar e distribuir cópias modificadas
- Consulte https://www.gnu.org/philosophy/free-sw.html
- O software de código aberto, de acordo com a definição da Iniciativa de código aberto, é tecnicamente semelhante ao Software Livre em termos de liberdades
  - https://www.opensource.org/docs/osd para obter a definição de Open SourceSoftware

Licenças Open-Source

- As licenças de software livre se enquadram em duas categorias principais
  - As licenças copyleft
  - As licenças não copyleft
- O conceito de copyleft é pedir reciprocidade nas liberdades concedidas ao usuário. O resultado é que quando você recebe um software sob uma licença de software copyleft free e distribuir versões modificadas dele, você deve fazê-lo sob a mesma licença
  - Mesmas liberdades para os novos usuários
  - É um incentivo para contribuir de volta com suas mudanças, em vez de mantê-las em segredo
- As licenças não copyleft não têm tais requisitos e as versões modificadas podem ser proprietárias, mas ainda requerem atribuição

Licenças Open-Source

- Algumas licenças CopyLeft
  - GPLv2/GPLv3 - General Public License, programas vinculados a uma biblioteca lançada sob a GPL também devem ser lançados sob a GPL, cuidado com GPLv3 em Sistemas Embarcados
  - LGPLv2/LGPLv3 - Lesser General Public License - Requisito linkagem dinâmica
- Algumas licenças Non-CopyLeft
  - Apache License
  - BSD License
  - MIT License
  - X11 License

IDE de Desenvolvimento e Depuração

IDE de Desenvolvimento

- VisualCode
  - Telemetry: O Visual Studio Code coleta dados de uso e os envia para a Microsoft. A Declaração de Privacidade da Microsoft refere-se a todos os produtos da Microsoft e não revela detalhes sobre essa coleta de dados.
  - Mas pode ser desabilitado nas configurações do usuário
- VsCodium
  - Uma compilação de código aberto 100% do Visual Studio Code, com a telemetria desativada
- Eclipse
- QtCreator

Depuração com GDB

- GDB - GNU Project Debugger, veja mais https://www.gnu.org/software/gdb/
- O GDB está disponível para a maioria das arquiteturas embarcadas
- Linguagens suportadas: C, C ++, Pascal, Objective-C, Fortran, Ada ...
- Interface do console (útil para depuração remota)
- Também pode ser usado por meio de IDEs gráficos

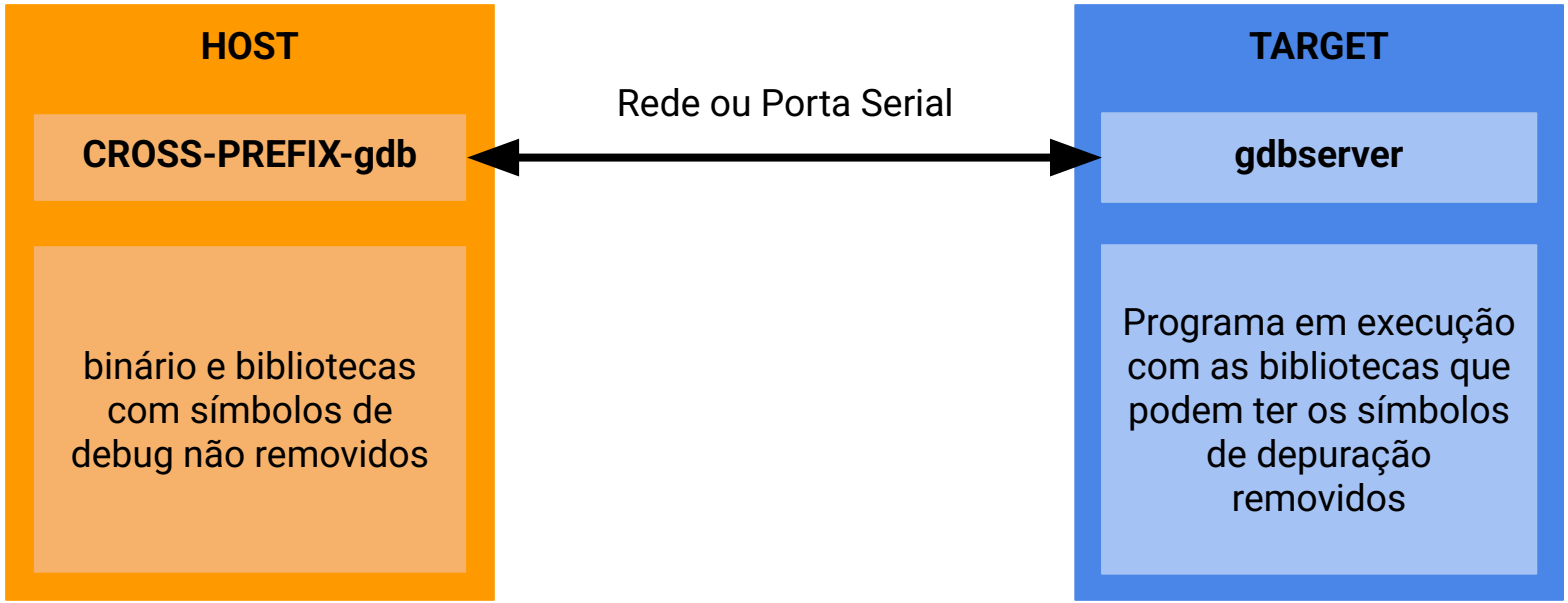
Depuração com GDB

- Pode ser usado para controlar a execução de um programa, definir pontos de interrupção ou alterar variáveis internas. Você também pode usá-lo para ver o que um programa estava fazendo quando travou (carregando sua imagem de memória, despejada em um arquivo central)

- GDBGui - Versão Gráfica desenvolvida em Python, repositório gdbgui

- Nova alternativa: lldb (https://lldb.llvm.org/) do projeto LLVM

Depuração com GDB - Debug Remoto



Depuração com GDB

- Opções de uso com GDB no **TARGET**
  - Serial

```
# gdbserver /dev/ttyS0 <programa> <parametros>
```

  - Rede

```
# gdbserver 0.0.0.0:<porta> <programa> <parametros>
```

  - Lançando em um processo em execução

```
# gdbserver --attach 0.0.0.0:<porta> <pid_programa>
```

Depuração com GDB

- Opções de uso com GDB no **HOST**
  - Serial

```
$ arm-unknown-linux-uclibcgnueabi-gdb
gdb> target remote /dev/ttyS0
```

  - Rede

```
$ arm-unknown-linux-uclibcgnueabi-gdb
(gdb) target remote <ip-addr>:<port>
```

Depuração com GDB

- Executando gdbserver no **TARGET**

```
# gdbserver 0.0.0.0:<porta> <programa> <parametros>
Process <programa> created; pid = 251
Listening on port <porta>
Remote debugging from host 192.168.254.153, port 56156
```

- Executando gdbserver no **HOST**

```
# arm-unknown-linux-uclibcgnueabi-gdb
(gdb) target remote 192.168.254.159:6677
Remote debugging using 192.168.254.159:6677
(gdb)
```

Outras ferramentas

- **valgrind** - Ferramenta para detectar BUGS de implementação com alocação de memória e **Helgrind** para erros e bugs com Threads

- **strace** - System Call Tracer, permite ver o que qualquer um de seus processos está fazendo: acessando arquivos, alocando memória, etc

- **ltrace** - Library Trace, uma ferramenta para rastrear chamadas de biblioteca usadas por um programa e todos os sinais que ele recebe

Laboratório

Prática 09

- Depurando Aplicações com GDB

Obrigado!

Cleiton Bueno - cleiton.bueno@b2open.com