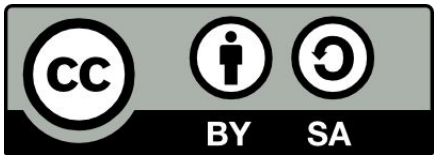
Yocto Project

Revisão: 03/03/2026

B2Open Systems

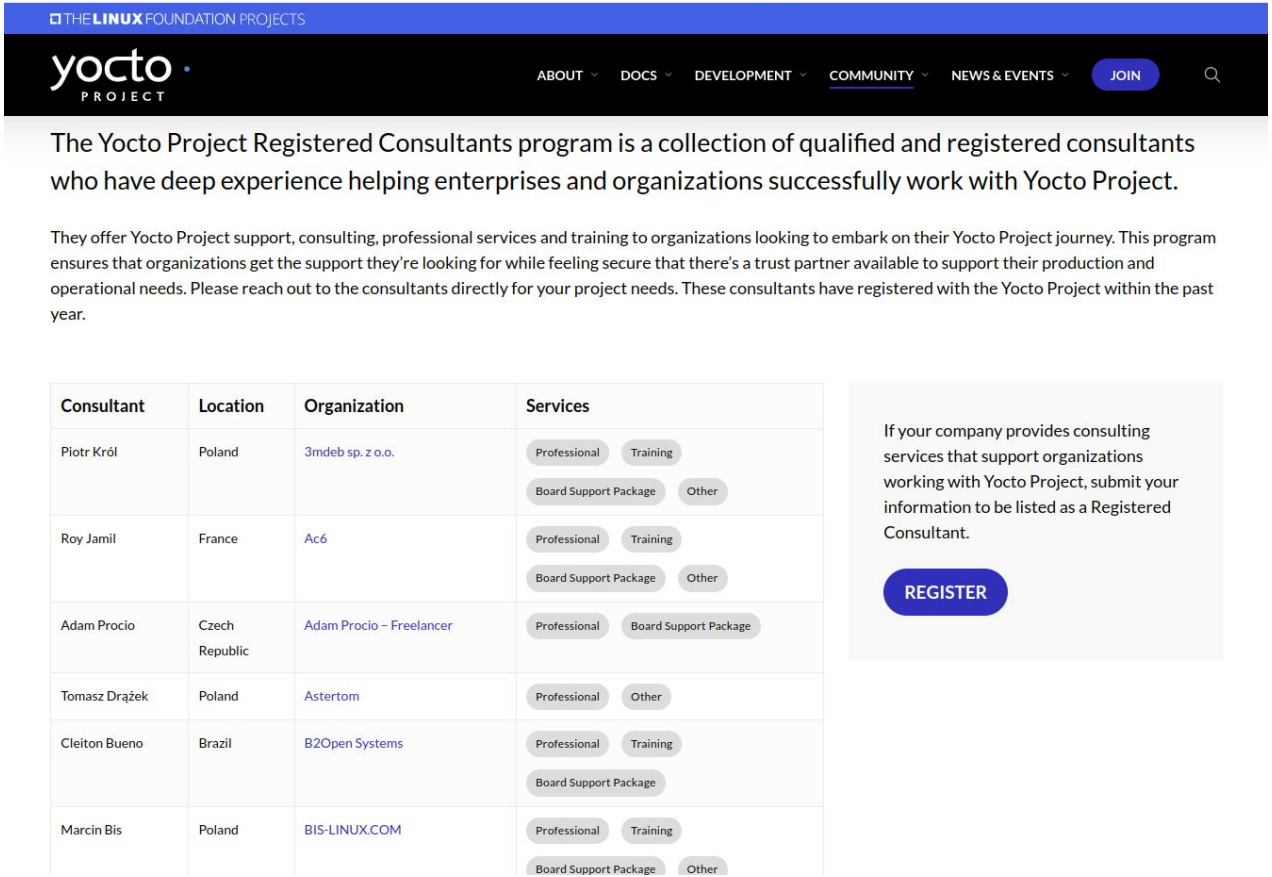


Informações Gerais

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

Cleiton Bueno



THE LINUX FOUNDATION PROJECTS

yocto PROJECT

ABOUT DOCS DEVELOPMENT COMMUNITY NEWS & EVENTS JOIN

The Yocto Project Registered Consultants program is a collection of qualified and registered consultants who have deep experience helping enterprises and organizations successfully work with Yocto Project.

They offer Yocto Project support, consulting, professional services and training to organizations looking to embark on their Yocto Project journey. This program ensures that organizations get the support they're looking for while feeling secure that there's a trust partner available to support their production and operational needs. Please reach out to the consultants directly for your project needs. These consultants have registered with the Yocto Project within the past year.

Consultant	Location	Organization	Services
Piotr Król	Poland	3mdeb sp. z o.o.	Professional, Training, Board Support Package, Other
Roy Jamil	France	Ac6	Professional, Training, Board Support Package, Other
Adam Procio	Czech Republic	Adam Procio – Freelancer	Professional, Board Support Package
Tomasz Drążek	Poland	Astertom	Professional, Other
Cleiton Bueno	Brazil	B2Open Systems	Professional, Training, Board Support Package
Marcin Bis	Poland	BIS-LINUX.COM	Professional, Training, Board Support Package, Other

If your company provides consulting services that support organizations working with Yocto Project, submit your information to be listed as a Registered Consultant.

REGISTER

Agenda

- **Dia 1** - Introdução Linux Embarcado, BuildSystem Yocto Project e primeiro build

- **Dia 2** - Estudo das layers e criando a própria camada, customizando DISTRO, MACHINE e IMAGE

- **Dia 3** - Criando Receitas, Receitas de Aplicações, Receitas com Serviços, Adicionando Usuário e considerações finais

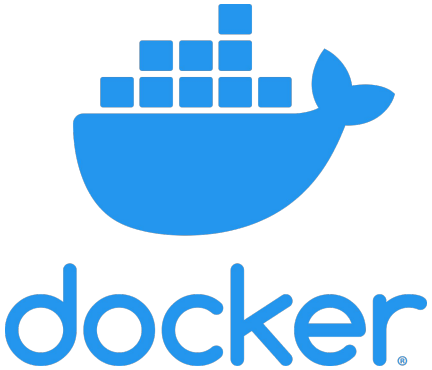
Material do Treinamento

SlidesYP.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

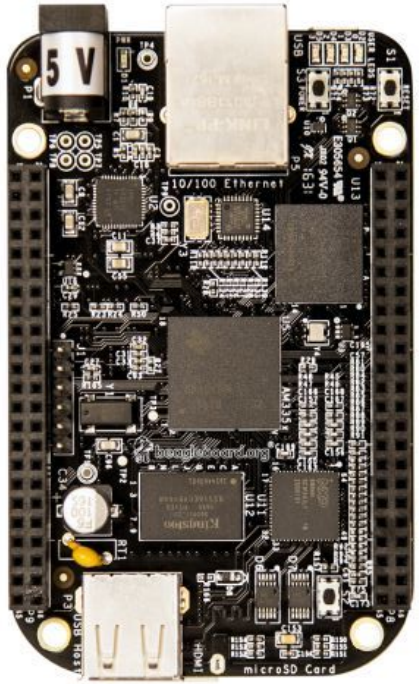


Participe!

- 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 - 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 - 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]

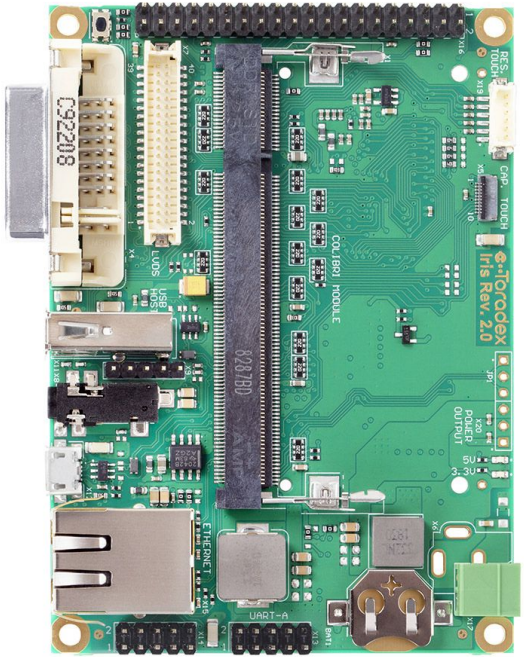
Hardware Utilizado - Toradex Colibri iMX8X

- 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 - Toradex BaseBoard IRIS

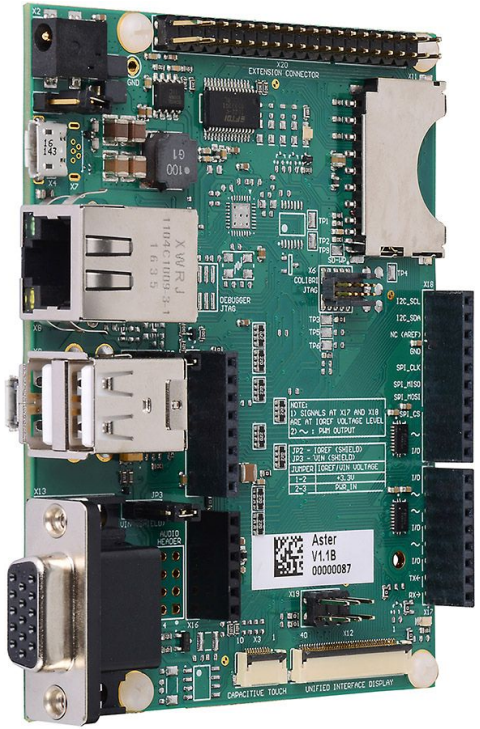
- Alimentação 6-27V DC
- 1x USB Host
- 1x USB OTG
- 3x UART's RS232
- 4x PWM
- 1x Ethernet
- 1x LVDS, 1x HDMI (Conector DVI), 1x VGA (Conector DVI)
- 1x RTC na placa
- 1x uSD
- >25 GPIO's



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 - 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]

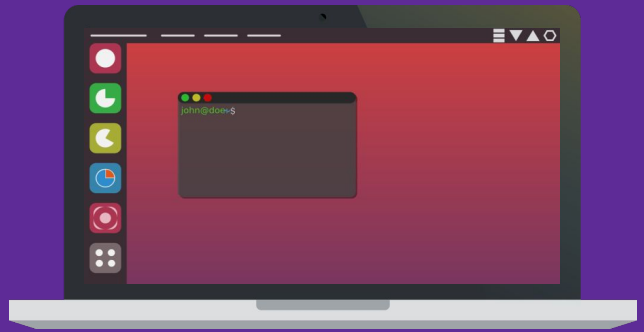
QEMU

- **QEMU** (Quick Emulator)
- É um software de virtualização de código aberto que permite a execução de sistemas operacionais e aplicativos em diferentes arquiteturas de hardware.
- Capaz de emular uma ampla variedade de processadores, como x86, ARM, PowerPC e SPARC.
- Emula o hardware necessário para executar um sistema operacional convidado, permitindo que ele seja executado em um ambiente isolado e seguro.
- QEMU também fornece recursos avançados, como emulação de dispositivos de entrada/saída, rede virtual e suporte a aceleração de hardware.
- Target padrão configurado no Yocto Project

Itens adicionais

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

Ambiente de Desenvolvimento



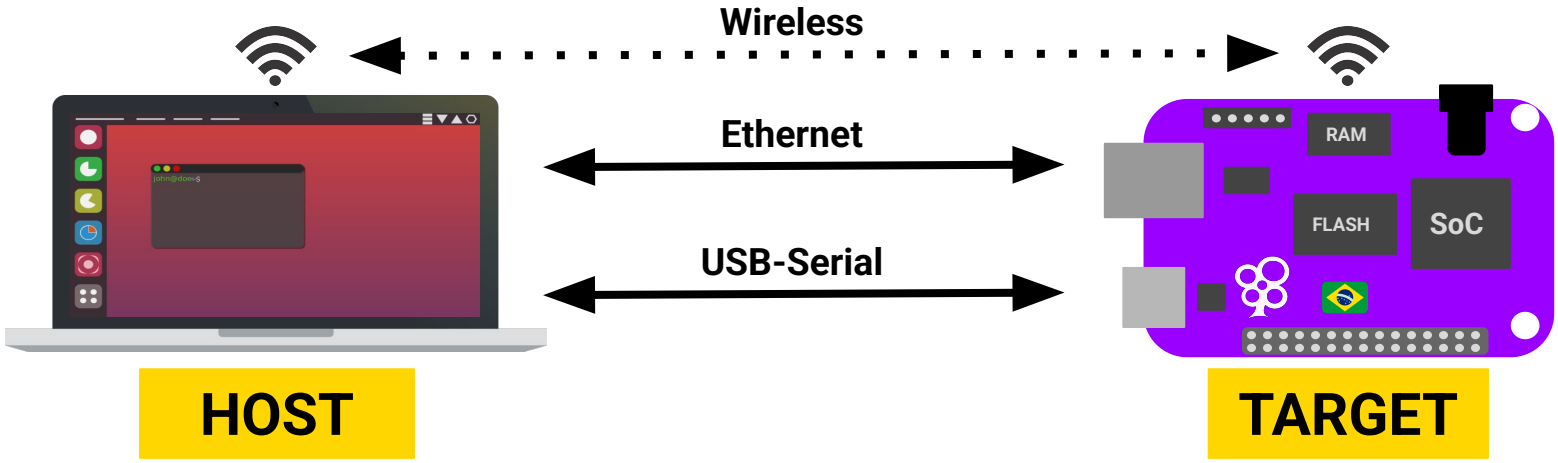
Ambiente de Desenvolvimento

- 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)

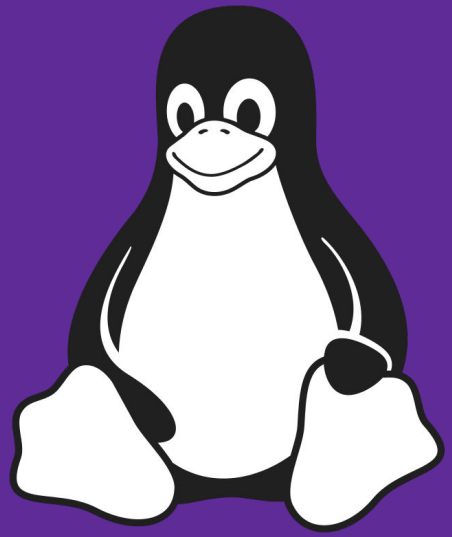
- Executando comandos root como usuário comum com **sudo**

Laboratório

Prática 01 e Prática 02

- Preparando o ambiente do Treinamento
- Prática Terminal e Comandos

Introdução ao Linux Embarcado



Linux Embarcado

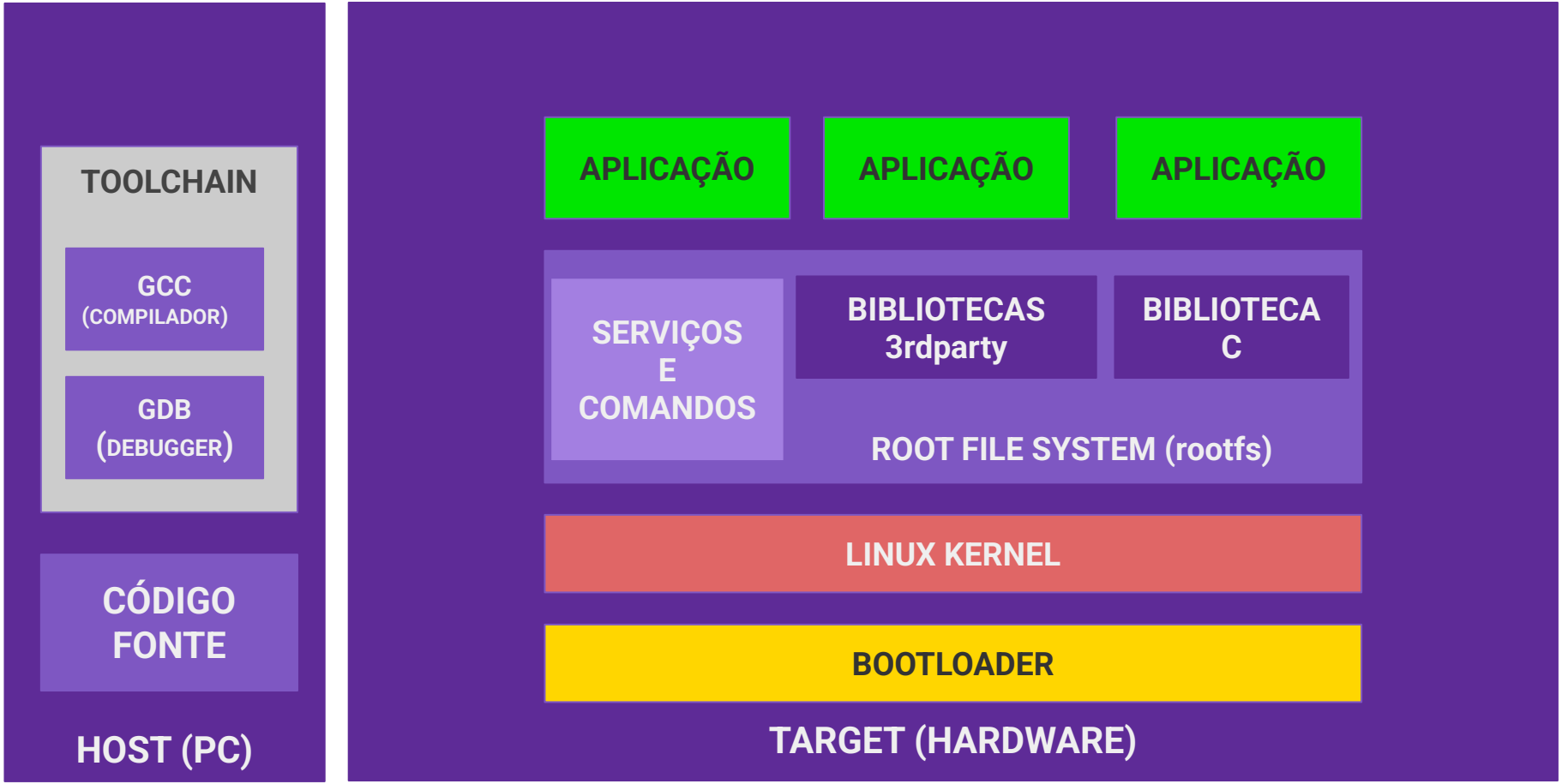
O que é Linux Embarcado?

Falar de Linux Embarcado é o mesmo de uma Distribuição Linux utilizada em Desktop ou Servidores, no entanto, na maioria das vezes com propósito bem definido e uma pilha de recursos resolvidas!

O que é Linux Embarcado?

Estamos falando do mesmo Kernel Linux, e muitas ferramentas em comuns como **binutils** mas em arquiteturas diferentes, como **ARM**.

Linux Embarcado



Linux Embarcado

Linux Embarcado

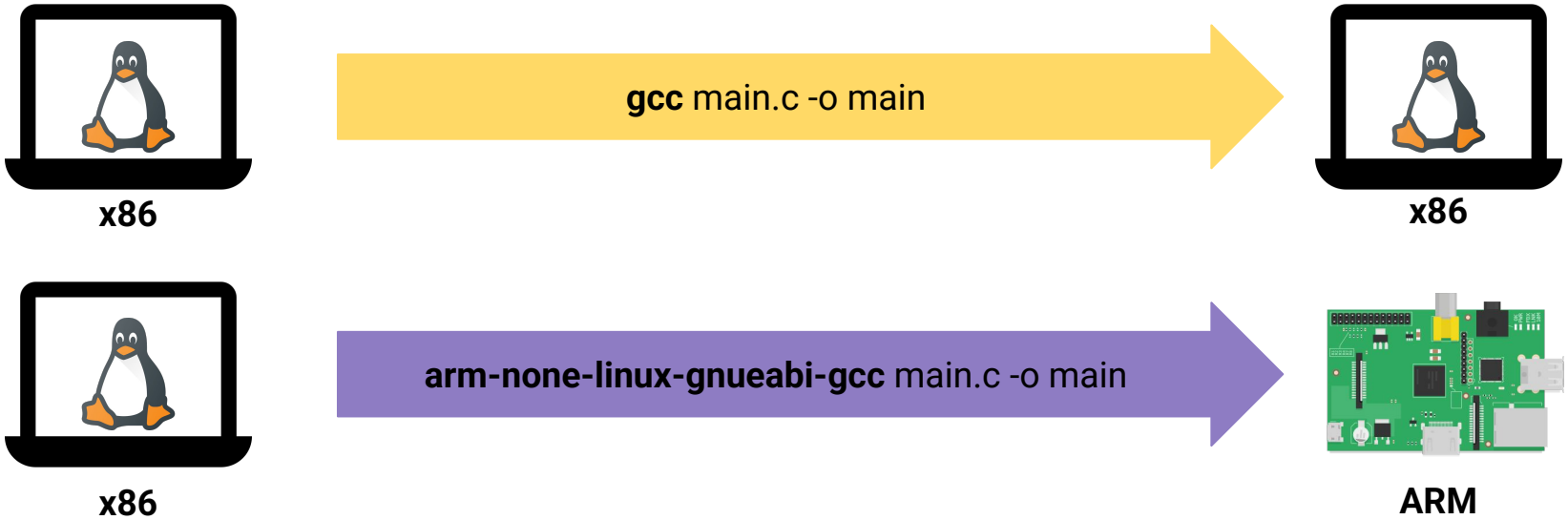
Sequência de inicialização

- **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.

Toolchain

- Conjunto de ferramentas que permitem compilar e gerar o binário final em **plataforma nativa** ou **compilação-cruzada**

- Necessário para gerar Bootloader, Kernel, Módulos do Kernel, Aplicações e Bibliotecas do RootFS



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)

Explorando Diferentes Abordagens de Criar um Linux Embarcado

- Distribuição Pronta
- Linux From Scratch
- Distribuição Customizada

Distribuição Pronta

- PRÓS
  - Sistema Operacional pronto para uso
  - Distribuições populares como Debian e Ubuntu - **Raspberry Pi OS**, **Armbian**
  - Pilha de desenvolvimento pronta ou de fácil preparação

- CONTRAS
  - Sistema Operacional maior, em sua maioria passando de 1G que implica diretamente em um maior tempo de inicialização
  - Customizações podem se tornar um pesadelo
  - Perda do controle de licenças e ferramentas instaladas
  - Difícil mapeamento de Riscos e Ameaças

Linux From Scratch - (Manual)

- PRÓS
  - Controle e abordagem manual do Toolchain até o binário final
  - Estudo e conhecimento absoluto por se envolver em todas etapas

- CONTRAS
  - É o método de geração de um Sistema Linux manualmente
  - Da preparação do ambiente (toolchain)
  - Até compilação de todos os componentes Bootloader, Kernel, Montar RootFS e adicionar o básico de ferramentas, exemplo: BusyBox
  - É um processo demorado, complexo e muitas vezes amarrado a versões/recursos/funcionalidades

Distribuição Customizada - Sistema de Build

- PRÓS
  - Existem excelentes opções open-source como: **Yocto Project**, **Buildroot**, **ELBE**, **Armbian** e **OpenWRT**
  - Há opções comerciais
  - Controle fiel de tudo que é instalado bem como versões e customizações
  - Gerar Sistemas menores e que implica em tempo de inicialização melhor
  - Controle fiel de licenças e restrições

- CONTRAS
  - Implicar em uma curva de aprendizagem e estudo da ferramenta
  - Importante verificar se o fornecedor do hardware utiliza alguma das opções Open-Source

Yocto Project

Um pouco de história

O **Yocto Project** é um conjunto de ferramentas projetado para auxiliar na criação de distribuições Linux personalizadas para dispositivos embarcados. Um esforço de diversos players da indústria de hardware e software para uma ferramenta de redução dos trabalhos ao construir um Linux Embarcado.

Entre os principais recursos está o **OpenEmbedded Core** como núcleo do Yocto Project, o OpenEmbedded foi criado em 2003 por alguns desenvolvedores do **OpenZaurus**, desde o início ele foi baseado/inspirado no escalonador de tarefas do Gentoo Portage conhecido por **Bitbake**.

A Distribuição **Poky** surgiu no projeto OpenEmbedded como uma referência de Distribuição estruturada e estável.

Em 2010 a Linux Foundation divulgou o **Yocto Project** e oficialmente lançado em 03.2011 baseado no OpenEmbedded e outras ferramentas em sua base.

Yocto Project

Vamos dar uma olhada em alguns dos projetos essenciais que fazem parte do Yocto Project:

- OpenEmbedded-Core
- BitBake
- Poky
- DevTool

Para mais informações sobre outros componentes do Yocto Project, visite https://www.yoctoproject.org/software-overview/project-components/.

Yocto Project - OpenEmbedded-Core

É o principal componente do Yocto Project, fornecendo uma estrutura central para a construção de sistemas Linux embarcados. Ele inclui um conjunto de **camadas** e **receitas** que ajudam a gerenciar a construção e a configuração do sistema operacional.

Yocto Project - BitBake

É a ferramenta central de construção utilizada pelo Yocto Project. É escrita em **Python**, faz parser das receitas e realiza o processo de construção, garantindo que todos os componentes sejam compilados corretamente e os pacotes sejam construídos de acordo com as especificações.

É um **escalonador de tarefas**, realiza processamento(*parser*) dos arquivos de receitas e começa a orquestrar a execução e suas dependências.

Yocto Project - Poky

É uma distribuição de referência do Yocto Project, fornecendo uma base sólida para a construção de sistemas Linux embarcados. Ela inclui uma seleção de pacotes, configurações e receitas pré-definidas, facilitando o início do desenvolvimento.

É uma referência para se utilizar em qualquer plataforma como BeagleBone Black e RaspberryPI, ou até mesmo emulado via **QEMU**.

Yocto Project

Um buildsystem como Yocto Project possui todos recursos necessários para gerar um ambiente **HOST** para o desenvolvedor e preparar o **TARGET** de acordo com suas necessidades e escopo do projeto.

O Yocto Project não é uma **Distribuição Linux**, mas possui todos recursos necessários para criar uma para você!

Por dentro do Yocto Project

Por debaixo do capô dos projetos Yocto Project

- **Metadados** - O Bitbake processa metadados, estes são divididos em:

  - **Configuração** (.conf) - Configurações globais que devem fornecer informações para as Classes e Receitas, como informações do Hardware e Recursos.

  - **Classes** (.bbclass) - Classes são herdadas por todo sistema e aplicam uma específica tarefa para todas receitas que o herdarem, evitando duplicidade de código/tarefas.

  - **Receitas** (.bb e .bbappend) - Receita irá descrever uma aplicação com rotinas de como e de onde baixar a aplicação, descompactar, configurar, compilar, “testar”, empacotar e instalar na imagem, pode ser feito com mistura de rotinas em ShellScript e Python.

Yocto Project

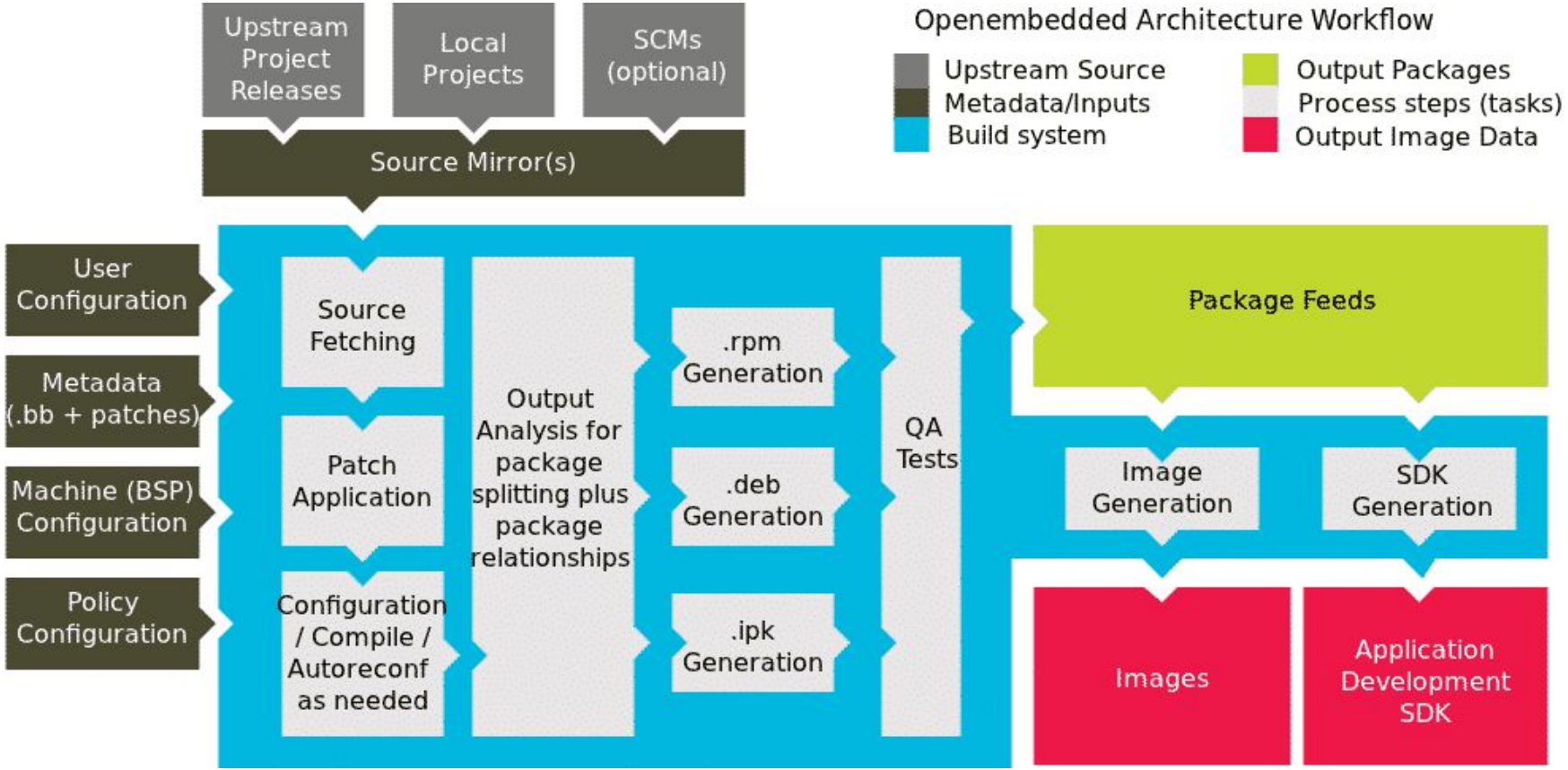
- **BSP** - Board Support Package camada com configurações e definições de como construir uma imagem para uma específica placa, no geral fornecido pelo fabricante(Bootloader, Kernel, Firmwares, Ferramentas, ...)

- **Distribuição** - Específica implementações para o Linux (Rootfs, Aplicações) - Variável **DISTRO**

- **Machine** - Definição de Arquitetura, Periféricos, Recursos e BSP(Kernel/Bootloader/Firmwares) - Variável **MACHINE**

- **Image** - Descrição da Imagem, detalhando o que deve ser instalado e configurações para a geração da Imagem

Fluxo de Trabalho OpenEmbedded BuildSystem



Yocto Project - Documentação

A documentação do Yocto Project é uma fonte valiosa de informações e orientações para os desenvolvedores que trabalham com o projeto. Ela fornece detalhes sobre os conceitos, componentes, ferramentas e processos envolvidos na construção de distribuições Linux personalizadas para dispositivos embarcados.

Existem várias fontes de documentação disponíveis para o Yocto Project. Aqui estão algumas:

- Yocto Project Technical Documentation
- Yocto Project Mega-Manual
- Yocto Project Development Manual

Yocto Project - Documentação

- **Yocto Project Technical Documentation**: Este é o portal oficial da documentação técnica do Yocto Project. Ele abrange uma ampla variedade de tópicos, desde conceitos básicos até informações avançadas sobre configuração, receitas, camadas, gerenciamento de pacotes, imagem do sistema, depuração e muito mais.

  https://www.yoctoproject.org/docs/

- **Yocto Project Mega-Manual**: O Mega-Manual é uma abrangente compilação de todas as seções da documentação técnica do Yocto Project. Ele oferece uma visão geral completa do projeto, permitindo uma exploração mais detalhada de cada tópico relevante.

  https://docs.yoctoproject.org/singleindex.html

- **Yocto Project Development Manual**: O Development Manual é direcionado para desenvolvedores que desejam criar e personalizar distribuições Linux usando o Yocto Project. Ele fornece instruções detalhadas sobre como criar receitas, adicionar pacotes, configurar imagens, gerenciar camadas, personalizar o ambiente de compilação e muito mais.

  https://www.yoctoproject.org/docs/current/dev-manual/

Yocto Project (Poky)

Obtendo o código-fonte e preparando o ambiente de construção

Pré-requisitos

Não é mandatório, mas evita vários problemas ao utilizar alguma Distribuição Linux tal como: Ubuntu, Debian, Fedora - Supported Linux Distributions.

Algumas dependências para correto funcionamento: gcc, git, python3, tar >1.4 - Lista completa das dependências em Required Packages for the Build Host.

```
$ apt update
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping
python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit
mesa-common-dev
```

Obtendo Poky

Para obter o código-fonte do Yocto Project, o processo é bastante simples. Basta utilizar o comando **git** para clonar o repositório do Poky.

O **Poky** é o componente central do Yocto Project e representa a distribuição de referência. É essencial especificar qual versão deseja clonar ao realizar o processo de clonagem.

Cada nova versão lançada pelo Yocto Project ocorre a cada 6 meses. Essas versões trazem melhorias significativas, novas funcionalidades e atualizações para as ferramentas, bibliotecas e recursos utilizados. Um exemplo disso é o GNU C Compiler (GCC), que pode ser atualizado nas versões mais recentes.

Portanto, é importante selecionar cuidadosamente a versão desejada ao clonar o Poky, garantindo assim acesso às melhorias e recursos mais recentes disponíveis no Yocto Project.

Releases

CODINOME	YP Version	Release Date	Estado Atual	Poky Version
Scarthgap	5.0	04/2024	LTS	2.8
Nanbield	4.3	11/2023	até 05/2024	2.6
Kirkstone	4.0	05/2022	LTS	2.0
Dunfell	3.1	04/2020	LTS	1.46

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

Baixando o Projeto

```
$ mkdir ~/yp
$ cd ~/yp
~/yp $ git clone -b scarthgap git://git.yoctoproject.org/poky.git
Cloning into 'poky'...
remote: Enumerating objects: 693613, done.
remote: Counting objects: 100% (1546/1546), done.
remote: Compressing objects: 100% (107/107), done.
remote: Total 693613 (delta 1474), reused 1478 (delta 1439), pack-reused 692067
Receiving objects: 100% (693613/693613), 217.15 MiB | 3.79 MiB/s, done.
Resolving deltas: 100% (504325/504325), done.
~/yp $
```

Baixando o Projeto

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

Estrutura do Projeto

- **bitbake** - Diretório com todas ferramentas do **bitbake**

- **documentation** - Diretório com toda **documentação completa** do Yocto Project e seus projetos, bitbake, desenvolvimento do kernel, desenvolvimento de BSP, profiling e tracing.

- **meta*** - Todo diretório iniciado com meta* é também conhecido como layer, onde cada layer tem uma responsabilidade em específico: **meta**(Metadados do OpenEmbedded-Core, base para utilizar QEMU), **meta-poky**(Metadados referência para Poky), **meta-yocto-bsp**(Metadados com BSP de referencias como Beaglebone) e **meta-skeleton**(Metadados com diversos exemplos de receitas para BSP, Kernel e Aplicações).

Estrutura do Projeto

- **scripts** - Diversos scripts como: **runqemu**(Executar uma imagem desenvolvida com QEMU), sstate-cache-management.sh(remover sstate duplicado), **recipestool** e **devtool**(criar e administrar receitas), **wic** entre outros.

- **oe-init-build-env** - Script que deverá sempre ser executado com **source** para preparar o ambiente(variáveis ambiente) para o processo de build e geração da imagem.

Criando um Projeto

Para iniciar um projeto no Yocto Project e preparar o ambiente no terminal, deve-se utilizar o script **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.

Criando um Projeto

```
~/yp/poky $ source oe-init-build-env build
This is the default build configuration for the Poky reference distribution.

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

You can now run 'bitbake <target>'

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

You can also run generated qemu images with a command like 'runqemu qemux86-64'.

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
```

Criando um Projeto

Após preparar o ambiente pela primeira vez, a seguinte estrutura será criada no diretório **build-yp/conf**.

```
~/yp/poky/build $ tree
.
└── conf
    ├── bblayers.conf
    ├── local.conf
    └── templateconf.cfg
```

Uma breve análise e configuração do **conf/local.conf** e **conf/bblayers.conf** antes de gerar a primeira imagem.

Arquivos de Configuração

- **local.conf** - Arquivo onde se configura o hardware alvo, configura Distribuição, Customizações do Hardware, Tipo de Empacotamento e até programas a serem instalados - em um projeto real deverá possuir poucas alterações, pois uma camada deve ser criada para toda customização.

- **bblayers.conf** - Arquivo onde serão adicionadas as camadas para extender suporte e recursos, por padrão, já inclui **meta**, **meta-poky** e **meta-yocto-bsp** - Um breve exemplo, para adicionar suporte ao **Framework Qt5**, deverá ser adicionada a camada **meta-qt5**.

- **templateconf.cfg** -Determina o local onde estão os arquivos conf referência para uso, padrão: **meta-poky/conf/**

Variáveis Ambiente

BUILDDIR	**Caminho absoluto para o diretório do projeto**
PATH	**Adicionado ao PATH do sistema scripts/ e bitbake/bin**
BB_ENV_EXTRAWHITE	**Define uma lista de variáveis de ambiente adicionais que devem ser preservadas e transferidas para o ambiente de compilação do BitBake**
BBPATH	**Usada para especificar os diretórios onde o BitBake procura por receitas, classes e outros componentes necessários para a compilação**

Ferramentas/Comandos

bitbake	**Principal comando utilizado para parser e executar as tarefas de (download, configurar, compilar, instalar, empacotar, ...)**
bitbake-*	**Comandos extras com bitbake**
oe-*	**Comandos openembedded-core**

Construindo a primeira imagem

Executando o bitbake se inicia o processo de parser das configurações, compilação e geração de imagem básica de acordo com o valor da variável **MACHINE**, valor padrão **qemux86-64**.

```
~/yp/poky/build $ bitbake core-image-minimal
```

```
~/yp/poky/build $ runqemu qemuarm

~/yp/poky/build $ runqemu qemuarm nographic
```

Construindo a primeira imagem

```
$ bitbake core-image-minimal

Loading cache: 100% |                                                                                | ETA:  --:--:--
Parsing recipes: 100% |##################################################| Time: 0:00:22
Parsing of 920 .bb files complete (0 cached, 920 parsed). 1877 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.8.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-20.04"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
DISTRO               = "poky"
DISTRO_VERSION       = "5.0.10"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = " "
meta
meta-poky
meta-yocto-bsp       = "scarthgap:10fba0085de5645bb0366dd309182e0532aeea82"
```

Resultado da construção

```
$ ls  -1 tmp/deploy/images/qemux86-64/

 bzImage
 core-image-minimal-qemux86-64.rootfs.ext4
 core-image-minimal-qemux86-64.rootfs.manifest
 core-image-minimal-qemux86-64.rootfs.qemuboot.conf
 core-image-minimal-qemux86-64.rootfs.spdx.tar.zst
 core-image-minimal-qemux86-64.rootfs.tar.bz2
 core-image-minimal-qemux86-64.rootfs.testdata.json
 modules-qemux86-64.tgz
```

Validando com QEMU

Utilizando a ferramenta **runqemu** pode-se validar o resultado da compilação/instalação de um software na imagem construída, realizando o boot via QEMU.

```
$ runqemu qemux86-64
```

Executando a imagem com QEMU desabilitando video console e utilizando apenas o terminal como console da imagem.

```
$ runqemu qemux86-64 nographic
```

Laboratório

Prática 03

- Instalando o Yocto Project
- Criando o primeiro projeto
- Compilando a primeira imagem
- Teste com runqemu

Bitbake

O maestro de uma orquestra

Bitbake

O BitBake funciona com base em receitas (recipes) que definem como construir e compilar os componentes do sistema. Essas receitas são escritas em uma linguagem específica do BitBake, que descreve as dependências, comandos e ações necessárias para criar um determinado componente ou pacote.

Aqui estão algumas opções de uso comuns do BitBake:

bitbake <recipe>: Esse é o comando básico para iniciar a construção de uma receita específica. Por exemplo, bitbake core-image-minimal inicia o processo de construção da imagem mínima do sistema.

bitbake -c <task> <recipe>: Esse comando executa uma tarefa específica para uma receita. As tarefas podem ser, por exemplo, fetch (obter o código-fonte), configure (configurar o ambiente de compilação), compile (compilar os componentes) ou install (instalar os pacotes no sistema de arquivos da imagem).

```
~/yp/poky/build $ bitbake core-image-minimal

~/yp/poky/build $ bitbake -c compile picocom
```

Bitbake

bitbake -s: Exibe um resumo das receitas disponíveis e respectivamente suas versões.

bitbake -e <recipe>: Mostra as variáveis de ambiente e suas configurações para uma receita específica. Isso é útil para verificar as configurações e personalizar o comportamento de uma receita.

bitbake -k <recipe>: Iŕa executar a receita ignorando os erros se ocorrer o máximo que puder.

bitbake -f -c <task> <recipe>: Esse comando executa uma tarefa específica para uma receita, com a opção -f irá executar mesmo que já tenha sido executada sem erros, irá executar novamente.

bitbake -c listtasks <recipe>: Esse comando lista todas as tarefas de uma receita

Bitbake

bitbake -c clean <recipe>: Essa opção é irá limpar os diretórios temporários e os arquivos gerados durante o processo de compilação da receita, mas preserva fontes de código-fonte, permitindo que você inicie uma nova compilação sem precisar baixar novamente o código-fonte.

bitbake -c cleansstate <recipe>: Essa opção realiza uma limpeza mais abrangente, além de remover os diretórios e arquivos temporários também apaga o cache de estado (sstate cache). O cache de estado é uma coleção de artefatos intermediários que são armazenados para evitar a necessidade de reconstruir certos componentes quando suas receitas não foram modificadas. A limpeza do sstate cache garante que todos os componentes sejam construídos do zero, garantindo um ambiente limpo e consistente para a compilação.

bitbake -c cleanall <recipe>: Essa opção é a mais abrangente das três, realiza uma limpeza completa de todos os diretórios, arquivos temporários e caches relacionados à receita especificada. Isso inclui a remoção do cache de estado, arquivos de log, registros, arquivos de configuração gerados e quaisquer outros arquivos que não são necessários para uma nova compilação.

Bitbake

bitbake -g <recipe>: Gera um gráfico de dependências para uma receita específica. Esse gráfico visualiza as dependências entre as receitas e ajuda a entender as relações entre os componentes.

```
~/yp/poky/build $ bitbake -g minicom
Loading cache: 100% |#########################################| Time: 0:00:00
Loaded 1644 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies
NOTE: PN build list saved to 'pn-buildlist'
NOTE: Task dependencies saved to 'task-depends.dot'


~/yp/poky/build $ bitbake -g minicom -u taskexp
```

Bitbake

Ao executar **bitbake <recipe>** a tarefa build é chamada para: listar, carregar e processar as tarefas na sequência correta.

Bitbake

Qual a ordem e os arquivos que o bitbake carrega e processa?

```
<build_dir>/conf/bblayers.conf
<layers>/conf/layer.conf
...
...
meta/conf/bitbake.conf
<build_dir>/conf/local.conf
```

Bitbake

Qual a ordem e os arquivos que o bitbake carrega e processa?

```
~/yp/poky/build $ bitbake -e
#
# INCLUDE HISTORY:
#
# /home/b2open/treinamento/yp/poky/build/conf/bblayers.conf
# /home/b2open/treinamento/yp/poky/meta/conf/layer.conf
# /home/b2open/treinamento/yp/poky/meta-poky/conf/layer.conf
# /home/b2open/treinamento/yp/poky/meta-yocto-bsp/conf/layer.conf
# /home/b2open/treinamento/yp/poky/meta-treinamento/conf/layer.conf
# conf/bitbake.conf includes:
#   /home/b2open/treinamento/yp/poky/meta/conf/abi_version.conf
#   conf/site.conf
#   conf/auto.conf
#   /home/b2open/treinamento/yp/poky/build/conf/local.conf
#   /home/b2open/treinamento/yp/poky/meta/conf/multiconfig/default.conf
#   /home/b2open/treinamento/yp/poky/meta/conf/machine/qemuarm.conf includes:
```

Camadas

O que são camadas e sua estrutura

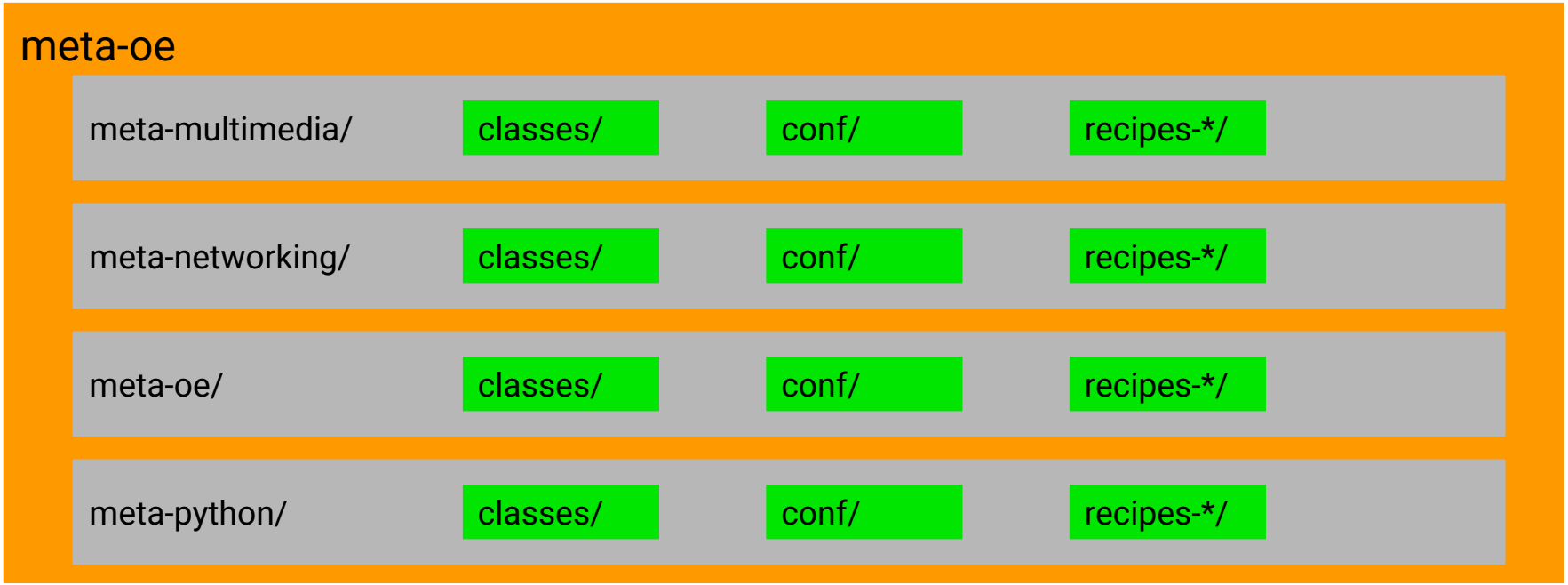
Estrutura das Camadas

- Novas camadas estendem novos recursos, funcionalidades, suporte a novos BSP e ferramentas, em sua maioria são repositórios git.

meta, meta-poky, meta-yocto-bsp	OpenEmbedded-Core + Yocto Project
meta-oe	OpenEmbedded Adicional
meta-qt5	Suporte Framework Qt5
meta-java	Suporte JAVA
meta-swupdate	SWUpdate
meta-raspberrypi	BSP RaspberryPI
meta-toradex-nxp	BSP Toradex*
meta-qcom	BSP Qualcomm
meta-ti	BSP TI

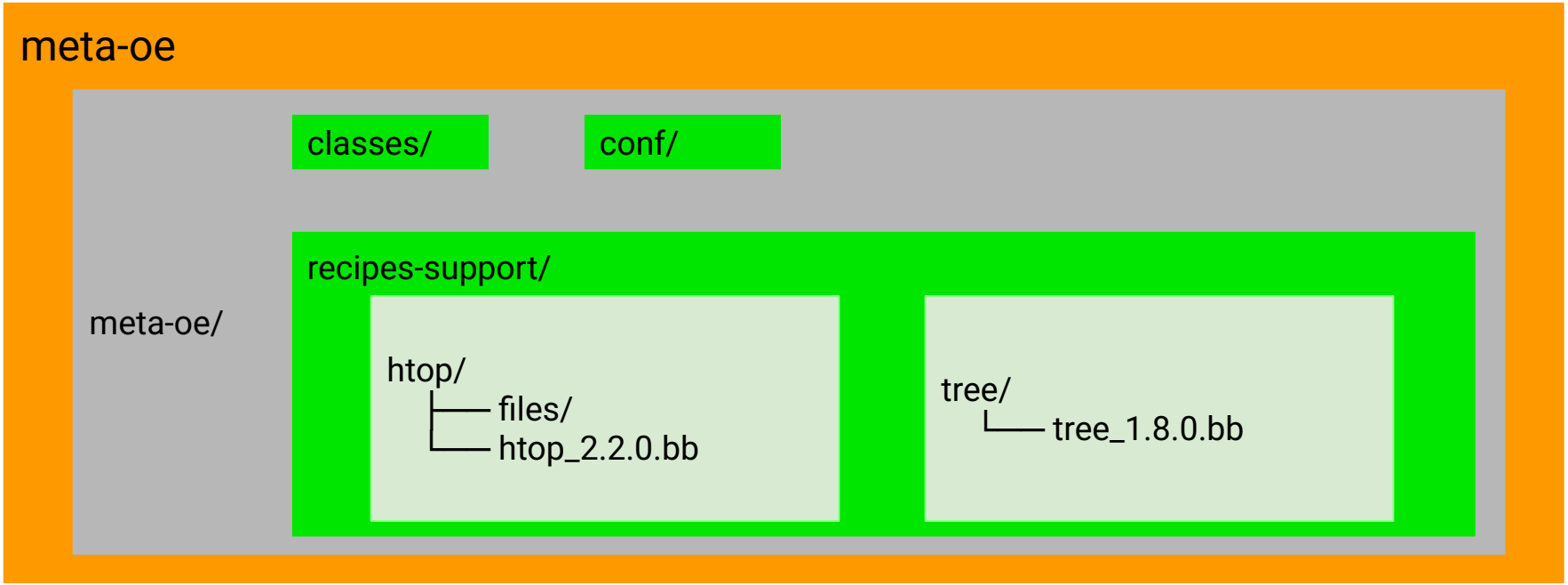
Estrutura das Camadas

- Novas camadas estendem os recursos, funcionalidades, suporte a novos BSP e ferramentas para construção do Projeto da imagem, em sua maioria são repositórios git.



Estrutura das Camadas

- Novas camadas estendem os recursos, funcionalidades, suporte a novos BSP e ferramentas para construção do Projeto da imagem, em sua maioria são repositórios git.



Estrutura das Camadas

meta-qt5/	classes/	qmake5.bbclass	Classe com tarefa comum para uso em várias receitas ou função comum
meta-qt5/	conf/	layer.conf	Em conf/ o layer.conf define configurações da camada meta-qt5, e conf/ onde se adiciona configurações de novas DISTRO e MACHINE
meta-qt5/	recipes-qt/*		Em recipes-* se adota os nomes dos diretórios das receitas e criar os arquivos de receitas .bb e as customizações com .bbappend

Estrutura das Camadas

meta-qt5/ recipes-qt/ qt5/ qtbase_git.bb

Uma receita (.bb)

meta-b2open/ recipes-qt/ qt5/ qtbase_git.bbappend

Uma customização da receita original qtbase_git.bbappend (.bbappend)

Criando sua própria camada

bitbake-layers

O **bitbake-layers** é uma interface de linha de comando que permite gerenciar as camadas do projeto. Algumas das funcionalidades oferecidas por essa ferramenta incluem:

Adicionar camadas, remover camadas, listar camadas e criar camadas.

Criando a própria camada

```
~/yp/poky/build $ bitbake-layers create-layer ../meta-b2open
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../meta-b2open'

~/yp/poky/build $ tree ../meta-b2open/
../meta-b2open/
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-example
    └── example
        └── example_0.1.bb
```

bblayers.conf

Utilizando o comando **bitbake-layers** para adicionar a camada criada ao build do projeto.

```
~/yp/poky/build $  bitbake-layers add-layer ../meta-b2open
NOTE: Starting bitbake server...
```

bblayers.conf

Utilizando o comando **bitbake-layers** para listar as camadas habilitadas no projeto.

```
~/yp/poky/build $  bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/b2open/treinamento/yp/poky/meta      5
meta-poky             /home/b2open/treinamento/yp/poky/meta-poky 5
meta-yocto-bsp        /home/b2open/treinamento/yp/poky/meta-yocto-bsp 5
meta-b2open           /home/b2open/treinamento/yp/poky/meta-b2open 6
```

Estrutura das Camadas

Analisando a configuração de uma camada que irá passar no parser do bitbake.

```
~/yp/poky $ cat meta-b2open/conf/layer.conf
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-b2open"
BBFILE_PATTERN_meta-b2open = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-b2open = "6"

LAYERDEPENDS_meta-b2open = "core"
LAYERSERIES_COMPAT_meta-b2open = "kirkstone scarthgap"
```

Estrutura das Camadas

Descrição das variáveis.

Variável	Descrição
BBFILES	O regex aplicado para encontrar as receitas .bb e .bbappend na camada
BBFILE_COLLECTIONS	Nome da camada para adicionar a coleção de layers que o bitbake procura por BBFILES
BBFILE_PRIORITY_<layer_name>	Prioridade da camada
LAYERDEPENDS_<layer_name>	Lista de camadas que são dependências
LAYERSERIES_COMPAT_<layer_name>	Lista das versões compatíveis com esta camada

Estrutura das Camadas

Uma camada de um projeto poderia conter uma estrutura como:

```
~/yp/poky $ tree -L 3 meta-b2open/
meta-b2open/
├── conf
│   └── layer.conf
├── recipes-b2open
│   └── ihm
│       └── ihm-rescovery_git.bb
├── recipes-core
│   └── images
│       └── b2open-core-image.bb
└── recipes-devtools
    └── gdb
        └── gdb_%.bbappend
```

local.conf

Configurando Variáveis para o primeiro build

local.conf

O arquivo local.conf criado em **<build-dir>/conf/local.conf** contém as principais configurações necessárias para geração de uma imagem para um hardware específico, algumas definições:

MACHINE - Configurar qual a plataforma alvo de Hardware, exemplo - verdin-imx8mp, verdin-am62, beaglebone-black, raspberrypi4 - padrão: **qemux86-64**

DISTRO - Distribuição Linux a ser gerada, padrão: **poky**

local.conf

DL_DIR - Diretório onde ficará armazenado todas ferramentas e códigos baixados para futuro reuso, faz download apenas uma vez, padrão (**<build-dir>/download**)

SSTATE_DIR - Diretório de **Shared State Cache** onde fica todos código configurado e compilado, e o Bitbake irá reaproveitar durante uma compilação para dependência, ou refazer a compilação do pacote, padrão (**<build-dir>/sstate-cache**)

TMP_DIR - Diretório temporário de trabalho durante todo processo de compilação de todo projeto e dependências - descompactar, configurar, compilar, empacotar e testar, padrão(**<build-dir>/tmp**)

local.conf

PACKAGES_CLASSES - Formato do empacotamento - package_rpm, package_deb ou package_ipk

EXTRA_IMAGE_FEATURES - Recursos e características que a IMAGE deverá possuir, exemplo: **ssh-server-openssh** - Adiciona suporte a SSH com OpenSSH

IMAGE_INSTALL - Lista com os nomes de pacotes a serem instalados na imagem.

local.conf

Uma configuração enxuta e minimalista do **<build-dir>/conf/local.conf**

```
~/yp/poky/build $ cat conf/local.conf
MACHINE ??= "qemuarm"

DL_DIR ?= "${TOPDIR}/downloads"

SSTATE_DIR ?= "${TOPDIR}/sstate-cache"

TMPDIR = "${TOPDIR}/tmp"

DISTRO ?= "poky"

PACKAGE_CLASSES ?= "package_rpm"

EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
```

bblayers.conf

Configurando Camadas

bblayers.conf

O arquivo **<build-dir>/conf/bblayers.conf** não há alterações além de adicionar novas camadas, uma dica use a variável **TOPDIR** para não deixar caminho relativo, evita problemas ao exportar o arquivo para outros computadores.

```
~/yp/poky/build $ cat conf/bblayers.conf
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  ${TOPDIR}/../meta \
  ${TOPDIR}/../meta-poky \
  ${TOPDIR}/../meta-yocto-bsp \
  ${TOPDIR}/../meta-b2open \
  "
```

bblayers.conf

Toda camada possui sua prioridade (**BBFILE_PRIORITY**), utilizado por exemplo quando existem duas receitas de mesma versão, a camada com maior prioridade pode definir a ser processada.

No caso de .bbappend, a camada com maior prioridade a receita será aplicada primeiro ao .bb original seguido dos demais .bbappend caso houver.

Laboratório

Prática 04

- Criando a própria camada
- Adicionado a camada ao projeto
- Práticas receitas com bitbake

Estrutura de diretórios do Build

Uma visão geral dos diretórios criados dinamicamente em <build-dir>

Estrutura de diretórios do build

Todo resultado do trabalho realizado estará dentro de **tmp** no diretório de build.

O resultado de configuração, patch, compilação, empacotamento, assim como a imagem final, kernel, bootloader e SDK/Toolchain.

Estrutura de diretórios do build

Diretório	Descrição
<build_dir>/tmp/buildstats	Informações detalhadas sobre cada etapa de compilação de todas receitas e tarefas
<build_dir>/tmp/cache	Cache dos metadados de todas receitas, Bitbake utiliza para agilizar futuras compilações
<build_dir>/tmp/deploy	Diretório com resultado final do trabalho: resumo das licenças, SDK, Imagens, Rootfs, Kernel, Bootloader, ...
<build_dir>/tmp/hosttools	Link-simbólico das ferramentas do Host que podem ser chamadas dentro da compilação
<build_dir>/tmp/log	Log de trabalho do Bitbake, separada das receitas
<build_dir>/tmp/sysroots	Bibliotecas e arquivos compartilhados entre receitas, após 3.1 sysroots está no WORKDIR das receitas
<build_dir>/tmp/work	Diretório com resultado de todo trabalho de cada receita para cada arquitetura

Estrutura de diretórios do build

O diretório **<build-dir>/tmp/work** é dividido por arquitetura, exemplo um build com **RaspberryPI3**.

Diretório	Descrição
all-poky-linux	Pacotes independentes de arquitetura, exemplo: tzdata
cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi	Pacotes dependentes da arquitetura de CPU(ARM), example: busybox e libstdc++
raspberrypi3-poky-linux-gnueabi	Pacotes dependentes do **MACHINE**, exemplo: Kernel e Bootloader
x86_64-linux	Conteúdo utilizado pelo **HOST** em sysroot nativo

Estrutura de diretórios do build

O diretório **<build-dir>/tmp/deploy**, exemplo um build com **RaspberryPI3**.

Diretório	Descrição
images	Contém a imagem final gerada bem como os componentes (Bootloader, Device-Tree, Kernel, Módulos do Kernel, RootFS e Image), separados por MACHINE. <build-dir>/tmp/deploy/images/${MACHINE}/
[deb, ipk ou rpm]	Contém todos os pacotes utilizadas na imagem ou para gerar como dependência, separado por arquitetura e as versões(-dev, -dbg, -doc, -src)
licenses	Contém todos os pacotes nativos e do TARGET e suas respectivas licenças e recipeinfo com nome, versão e licença de cada pacote
sdk	Diretorio onde é gerado um .sh instalado do SDK/Toolchain criado pelo Bitbake bem como manifesto do .sh gerado

Variáveis

Definição de Variáveis dos metadados

Variáveis - Atribuições

As variáveis que o BitBake realiza parser segue o formato **CHAVE = VALOR**, CHAVE sempre um nome em **MAIÚSCULO**.

Para acessar o conteúdo de uma variável deve-se utilizar ${CHAVE} e não $CHAVE.

Sequência de valores deve ser separado por espaço: **CHAVE = " 01 02 03"**

Variáveis - Atribuições

Diretório	Descrição
VARIAVEL = "arm"	Atribui o valor ABC para VARIAVEL no momento que encontrar a VARIAVEL no parsing
VARIAVEL ?= "arm64"	Atribui o valor ABC para VARIAVEL no momento que encontrar a VARIAVEL no parsing, caso ainda não esteja atribuída
VARIAVEL ??= "x86"	Atribui o valor ABC para VARIAVEL no final parsing, caso ainda não esteja atribuída
VAR2 = "${VARIAVEL} sec"	O conteúdo de VARIAVEL será expandido no momento que VAR2 é chamado para uso
VAR3 := "${VARIAVEL} gui"	A atribuição e expansão das variáveis ocorre no momento do parsing e não quando chamar VAR3

Variáveis - Atribuições

Diretório	Descrição
VARIAVEL_append = " m4"	Adiciona um valor para VARIAVEL, incluir espaço antes do VALOR
VARIAVEL_prepend = "qemu"	Adiciona o valor antes do conteudo da VARIAVEL e na sequência um espaço
VARIAVEL_remove = "m7"	Remove um valor do conteudo da variavel
VARIAVEL += "riscv" VARIAVEL =+ "m0"	Adiciona um valor para VARIAVEL, já inclui o espaço antes/depois do VALOR
VARIAVEL -= "x86"	Remove o valor da VARIAVEL
VARIAVEL .= "arm" VARIAVEL =. "arm"	Atribuição de ABC sem adicionar espaço antes ou depois

Variáveis - Atribuições <nova sintaxe>

Diretório	Descrição
VARIAVEL:append = " m4"	Adiciona um valor para VARIAVEL, incluir espaço antes do VALOR
VARIAVEL:prepend = "qemu"	Adiciona o valor antes do conteudo da VARIAVEL e na sequência um espaço
VARIAVEL:remove = "m7"	Remove um valor do conteudo da variavel

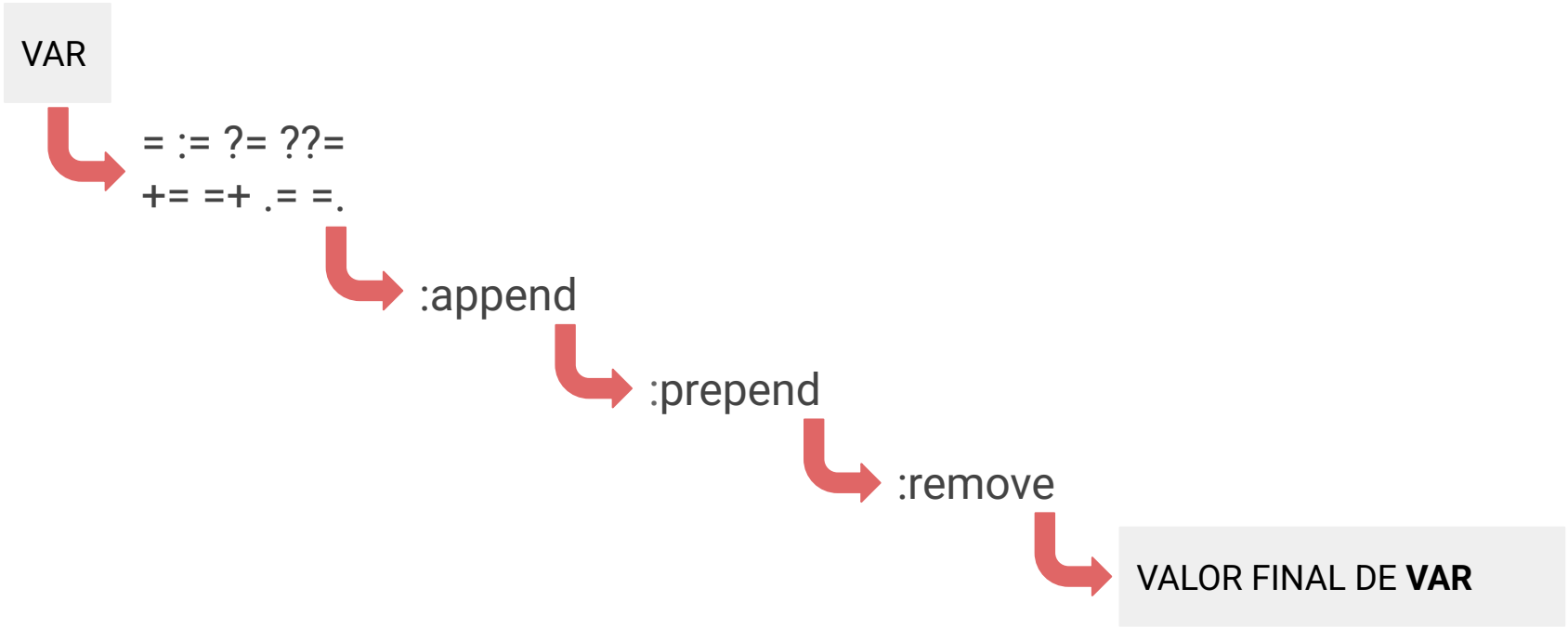
Variáveis - Atribuições

Para utilizar o valor de uma variável em outra variável, para expandir o conteúdo utiliza-se o operador **"${ }"**, exemplo:

```
VARIAVEL = "IPV6"
VARIAVEL_FEATURES = "${VARIAVEL} FASTETHERNET"

VARIAVEL_FEATURES:append = " IPV4"
VARIAVEL_FEATURES += "MAC"

VARIAVEL_FEATURES:remove = "IPV6"
```

Valor de **VARIAVEL_FEATURES** -> **"FASTETHERNET IPV4 MAC"**

Ordem de Atribuição



OVERRIDES

Expandindo valores de forma condicional

OVERRIDES

O BitBake usa **OVERRIDES** para controlar quais variáveis são substituídas após o BitBake analisar receitas e arquivos de configuração.

O valor padrão de OVERRIDES inclui os valores das variáveis **CLASSOVERRIDE**, **MACHINEOVERRIDES** e **DISTROOVERRIDES**, os valores são separados por ‘:’.

Antes do Honister, a sintaxe para **OVERRIDES** usava _ em vez de ‘:’.

OVERRIDES

Utilizando **OVERRIDES** uma receita pode processar, configurar e aplicar de acordo com a configuração de OVERRIDES em DISTRO ou MACHINE, exemplo:

```
OVERRIDES:append = ":black-box"
```

```
LEVEL_SECURITY = "0"
LEVEL_SECURITY:rpi = "1"
LEVEL_SECURITY:qemux86 = "0"
LEVEL_SECURITY:qemuarm = "1"
LEVEL_SECURITY:black-box = "3"
```

Laboratório

Prática 05

- Prática com Variáveis
- Prática com OVERRIDES
- Prática com atribuições e Variáveis

Receitas

Anatomia de uma receita

Receitas

Uma receita (.bb) ou uma modificação/atualização (.bbappend) é um arquivo legível com diversas variáveis com propósitos de:

- Descrever sobre o pacote da receita
- Tipo de Licença
- URL para obter o código da receita
- Buildsystem a ser utilizado: Autotools, CMake, Scons, SetupTools, ...
- Customização de Flags para compilação
- Customização de instalação do pacote

Receitas -> Pacotes

Ingredientes + Ferramentas
(recipes + conf + bitbake)

Bolo
(pacote)

Receitas -> Pacotes

```
SUMMARY = "Lightweight and minimal dumb-terminal emulation program"
SECTION = "console/utils"
LICENSE = "GPL-2.0-or-later"
HOMEPAGE = "https://github.com/npat-efault/picocom"
LIC_FILES_CHKSUM =
"file://LICENSE.txt;md5=3000e4830620e310fe65c0eb69df9e8a"

BASEPV = "3.1"
PV = "${BASEPV}+git${SRCPV}"

SRCREV = "90385aabe2b51f39fa130627d46b377569f82d4a"

SRC_URI = "git://github.com/npat-efault/picocom;branch=master;protocol=https"

S = "${WORKDIR}/git"

EXTRA_OEMAKE = '"CC=${CC}' 'LD=${CC}' 'VERSION=${BASEPV}' \
                           'CFLAGS=${CFLAGS}' 'LDFLAGS=${LDFLAGS}' "

do_install () {
   install -d ${D}${bindir}
   install -m 0755 ${BPN} pcasc pcxm pcym pczm ${D}${bindir}/
}
```

picocom

(binário)

picocom

(pacotes de instalação deb, rpm ou ipk)

Ingredientes + Ferramentas
(recipes + conf + bitbake)

picocom
(pacotes)

Nome de um Receita

${PN}_${PV}.bb

meta/recipes-connectivity/iw/ iw_6.7.bb

Package Name: iw
Package Version: 6.7

Dentro da receita é esperada uma URL com esta versão 6.7 para download.

meta/recipes-devtools/mmc/ mmc-utils_git.bb

Package Name: mmc-utils
Package Version: git

Dentro da receita é esperada uma URL para algum repositório git, e na receita a especificação do nome do branch e revision(commit).

Compilando uma receita

Compilando as receitas do slide anterior:

```
~/yp/poky/build $ bitbake iw

~/yp/poky/build $ bitbake mmc-utils
```

Compilando uma receita

Ao final do processo os pacotes serão criados de acordo com a variável **PACKAGE_CLASSES** no conf/local.conf.

```
~/yp/poky/build $ ls -1 tmp/deploy/rpm/core2_64/iw*
tmp/deploy/rpm/core2_64/iw-6.7-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/iw-dbg-6.7-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/iw-dev-6.7-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/iw-doc-6.7-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/iw-src-6.7-r0.core2_64.rpm


~/yp/poky/build $ ls -1 tmp/deploy/rpm/core2_64/mmc-utils-*
tmp/deploy/rpm/core2_64/mmc-utils-0.1+git0+b5ca140312-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/mmc-utils-dbg-0.1+git0+b5ca140312-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/mmc-utils-dev-0.1+git0+b5ca140312-r0.core2_64.rpm
tmp/deploy/rpm/core2_64/mmc-utils-src-0.1+git0+b5ca140312-r0.core2_64.rpm
```

Pacotes da Receita iw

iw_6.7.bb

Pacote	Descrição
iw_6.7-r0_*.<PKG>	Contém o arquivo binário do programa, e pode conter arquivos de configuração e bibliotecas a serem instalados
iw-dbg_6.7-r0-*.<PKG>	Contém arquivos para depurar o software e geralmente contém símbolos.
iw-dev_6.7-r0_*.<PKG>	Contém os arquivos necessários para compilação para outras receitas (*.h, *.cmake)
iw-doc_6.7-r0_*.<PKG>	Contém documentação e arquivos de manuais do software
iw-src_6.7-r0_*.<PKG>	Contém os arquivos do código-fonte
iw-staticdev_6.7-r0_*.<PKG>	Contém arquivos de bibliotecas estáticas (*.a)

Explorando Variáveis

Variável	Característica/Função
PN	Nome da receita - A receita git_1.0.bb o PN é git
PV	Versão da receita - A receita git_1.0.bb o PV é 1.0
PR	Revisão da receita - A receita git_1.0.bb o PR é definido dentro da receita, padrão começa em r0
WORKDIR	Diretório de trabalho da receita: fetch, unpack, patch, compile, ...
BPN	Nome da receita com remoção de sufixos como nativesdk-, -native, -cross, ...

Explorando Variáveis

Variável	Característica/Função
S	Diretório onde é extraído o código-fonte da receita, segue o padrão - **${WORKDIR}/${PN}-${PV}** ou **${WORKDIR}/git**
D	Diretório de instalação da receita após compilação e antes de empacotar, segue padrão - **${WORKDIR}/image**
B	Diretório de build da receita **${WORKDIR}/build**
T	Diretório **${WORKDIR}/temp** onde será gerado todos os logs de trabalho da receita
THISDIR	Diretório relatório onde esta a receita e seus arquivos
DEFAULT_PREFERENCE	É usada para especificar a preferência padrão dessa receita em relação a outras receitas que fornecem o mesmo pacote. Use **-1** para “ignorar” a versão da receita.

Explorando Variáveis - Específico em Receitas

Variável	Característica/Função
SUMMARY DESCRIPTION	Descrição sobre a receita o conteúdo da receita
DEPENDS	Dependências (outras receitas) necessárias para compilação da receita
RDEPENDS	Dependências para “runtime” da aplicação no TARGET
SRC_URI	URL de https, git, svn, ftp, file entre outros para baixar o código-fonte da receita
LICENSE	Especificar o tipo de licença da receita da aplicação, lista de licenças.
LIC_FILES_CHKSUM	Arquivo da licença seguido de um hash para anexar ao pacote

Explorando Variáveis - Específico em Receitas

Variável	Característica/Função
SRCREV	Caso o SRC_URI seja um repositório Git, obrigatoriamente deverá especificar a revision ou melhor o commit a usar como referência na cópia
SRC_URI[sha256sum]	Caso o SRC_URI seja um link com um compactado(.zip, .tar.*) deverá especificar o valor de checksum o arquivo baixado
FILES:${PN}	A lista de arquivos e diretórios a criar/adicionar em um pacote de uma receita.
SYSTEMD_SERVICE:${PN}	Adicionar o arquivo <nome>.service para ser instalado e configurado para iniciar a aplicação no TARGET

Explorando Variáveis - Configuração Extra BuildSystem

BuildSystems	Variável
Autotools	EXTRA_OECONF
Cargo	EXTRA_OECARGO
CMake	EXTRA_OECMAKE
Make	EXTRA_OEMAKE
Meson	EXTRA_OEMESON
NPM	EXTRA_OENPM
SCons	EXTRA_OESCONS
WAF	EXTRA_OEWAF

Explorando Variáveis - Configuração Extra BuildSystem

Receita FFMPEG 6.1.2

Receita PulseAudio

Receita NanoMSG 1.2.1

ESTRUTURA DE UMA RECEITA

Criando uma receita básica

Estrutura de uma receita

Um exemplo de estrutura da receita **b2open-parameters_1.0.bb**:

```
DESCRIPTION = "B2Open Operating Parameters"
HOMEPAGE = "https://www.b2open.com"
LICENSE = "CLOSED"

SRC_URI = " \
                file://b2-params.conf \
                file://b2-params.sh \
                "


do_install() {
    install -d ${D}${bindir}
    install -d ${D}${sysconfdir}/b2open
    install -m 0755 b2-params.sh ${D}${bindir}
    install -m 0660 b2-params.conf ${D}${sysconfdir}/b2open
}
```

Estrutura de uma receita

Um exemplo de gerar pacote da receita **b2open-parameters_1.0.bb**:

```
~/yp/poky/build $ bitbake b2open-parameters
```

Receita - SRC_URI

GIT

```
SRC_URI = "git://<url>;protocol=<protocol>;branch=<branch>"
SRCREV = "<commit_hash>"
```

https, http e ftp

```
SRC_URI = "https://<url>/<PN>-<PV>.tar.gz"
SRC_URI[md5sum] = "<HASH MD5>"
SRC_URI[sha256sum] = "<HASH SHA256>"
```

Referência: bitbake-user-fetching#fetchers

Receita - FILESPATH

A variável **FILESPATH** é uma variável de ambiente que define os caminhos onde o **bitbake** procura por arquivos necessários durante o processo de compilação de um pacote. Esses arquivos podem incluir patches, arquivos de configuração e outros recursos necessários para construir o pacote corretamente.

A variável **FILESPATH** é uma lista de diretórios que devem estar separados por dois-pontos (:).

Receita - FILESPATH

No caso abaixo adicionando um arquivo local via **SRC_URI**:

```
SRC_URI += "file://defconfig"
```

O Bitbake irá procurar pelas combinações:

```
meta-<camada>/recipes-kernel/linux/files
meta-<camada>/recipes-kernel/linux/${BP}
meta-<camada>/recipes-kernel/linux/${BPN}
meta-<camada>/recipes-kernel/linux/files/${OVERRIDES}
meta-<camada>/recipes-kernel/linux/${BP}/${OVERRIDES}
meta-<camada>/recipes-kernel/linux/${BPN}/${OVERRIDES}
```

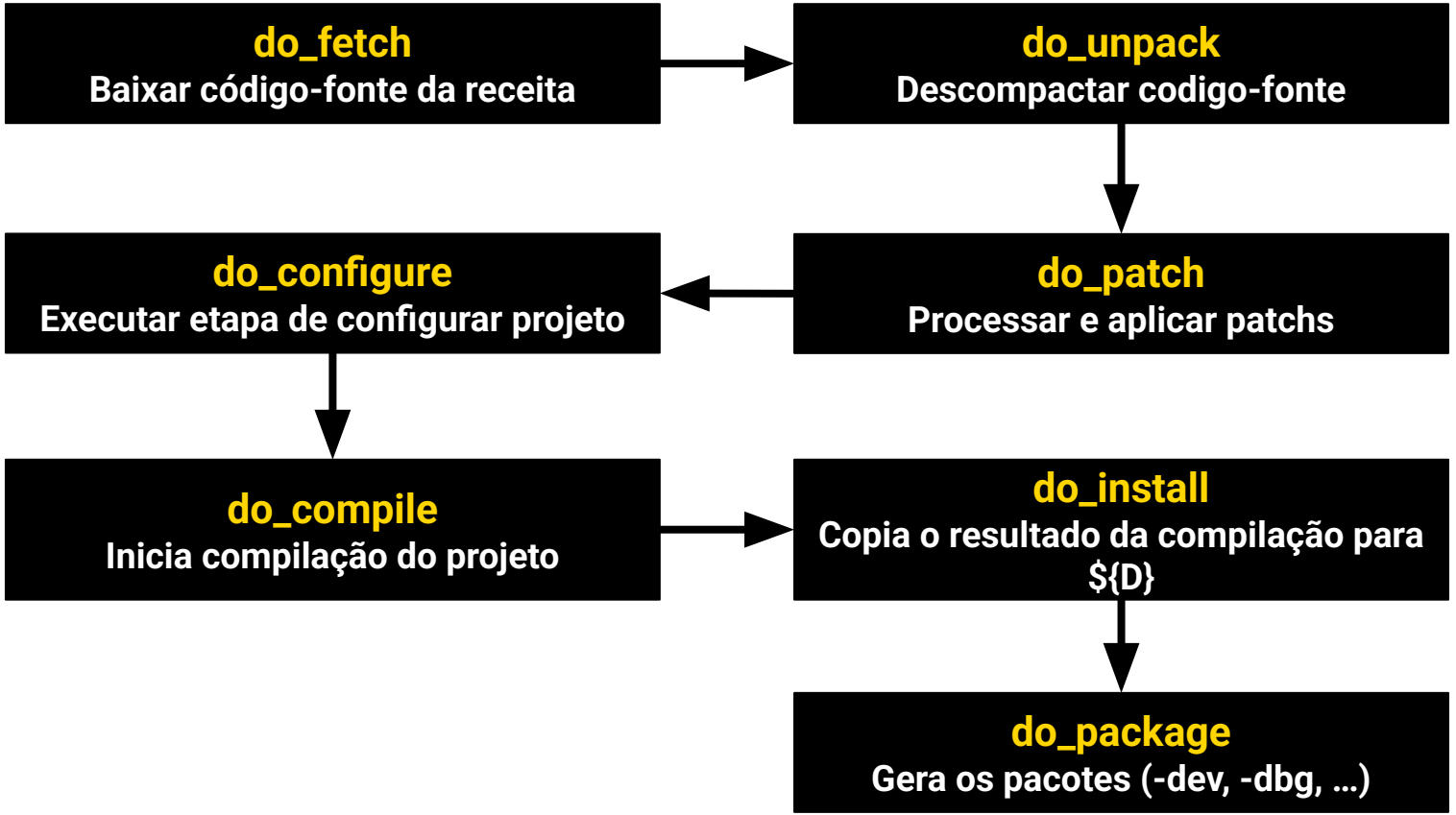
Receita - SRC_URI

Classe	Descrição
Github	SRC_URI "git://github.com/siposcsaba89/socketcan-cpp.git;protocol=git;branch=master" SRCREV = "5e328b7afe70b359a06f340ac7240ae36a544f5c"
Bitbucket	SRC_URI = "git://git@bitbucket.org/<company>/<NOME>.git;protocol=ssh;branch=master"
Arquivos Locais	SRC_URI = "file://ihm.service"
URL	SRC_URI = "https://gstreamer.freedesktop.org/src/gst-libav/gst-libav-${PV}.tar.xz" SRC_URI[md5sum] = "value" SRC_URI[sha256sum] = "value"
Git SubModule	SRC_URI="gitsm://git@github.com/<GROUP>/<PROJ_NAME>.git;protocol=ssh"
Git TAG	SRC_URI = "git://www.denx.de/git/u-boot.git;protocol=git;tag=${TAG}"

Receita - Lista de Tarefas

Tarefa	Descrição
do_fetch	Através da URL apontado em **SRC_URI** irá fazer download do código-fonte e armazenar em **DL_DIR**
do_unpack	Após o download, irá descompactar o conteúdo no **WORKDIR** do diretório de trabalho da receita
do_patch	Aplica algum patch caso necessário e especificado em **SRC_URI**
do_configure	Realiza pré-configuração antes de compilar, caso necessário.
do_compile	Compilação do código-fonte utilizando - GNU Autotools, CMake, SCons, ...
do_install	Copia o resultado da compilação de **B**(diretório de build) para **D**(diretório de destino)
do_populate_sysroot	Popula os artefatos de compilação da receita para sysroot para uso em outras receitas
do_package	Cria o pacote como resultado da compilação, em um dos formatos: DEB, RPM ou OPKG

Receita - Sequência das Tarefas



Receita - Ignorando uma tarefa em específico

Às vezes é necessário desabilitar uma das tarefas devido a uma particularidade do projeto.

```
do_configure[noexec] = "1"
```

ou

```
do_configure() {
    :
}
```

Receita - Expandindo Tarefa

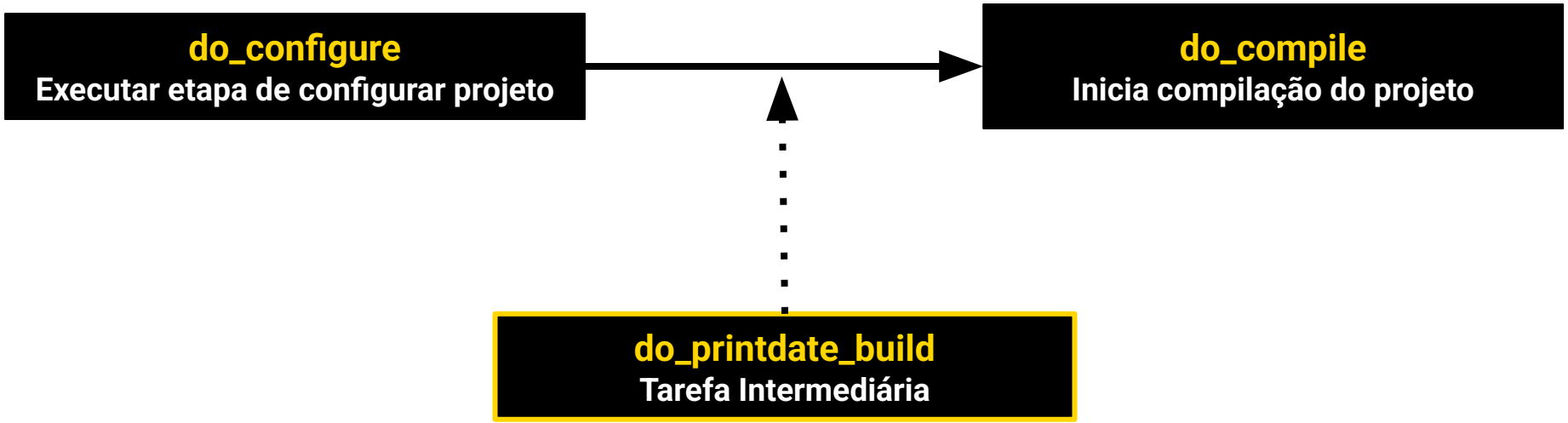
Sintaxe antiga (antes de Kirkstone [4.0]):

```
do_install_append() {
        install -m 0755 ${WORKDIR}/app.conf  ${D}${sysconfdir}/app.conf
}
```

Sintaxe atual:

```
do_install:append() {
        install -m 0755 ${WORKDIR}/app.conf  ${D}${sysconfdir}/app.conf
}
```

Receita - Adicionando Tarefas



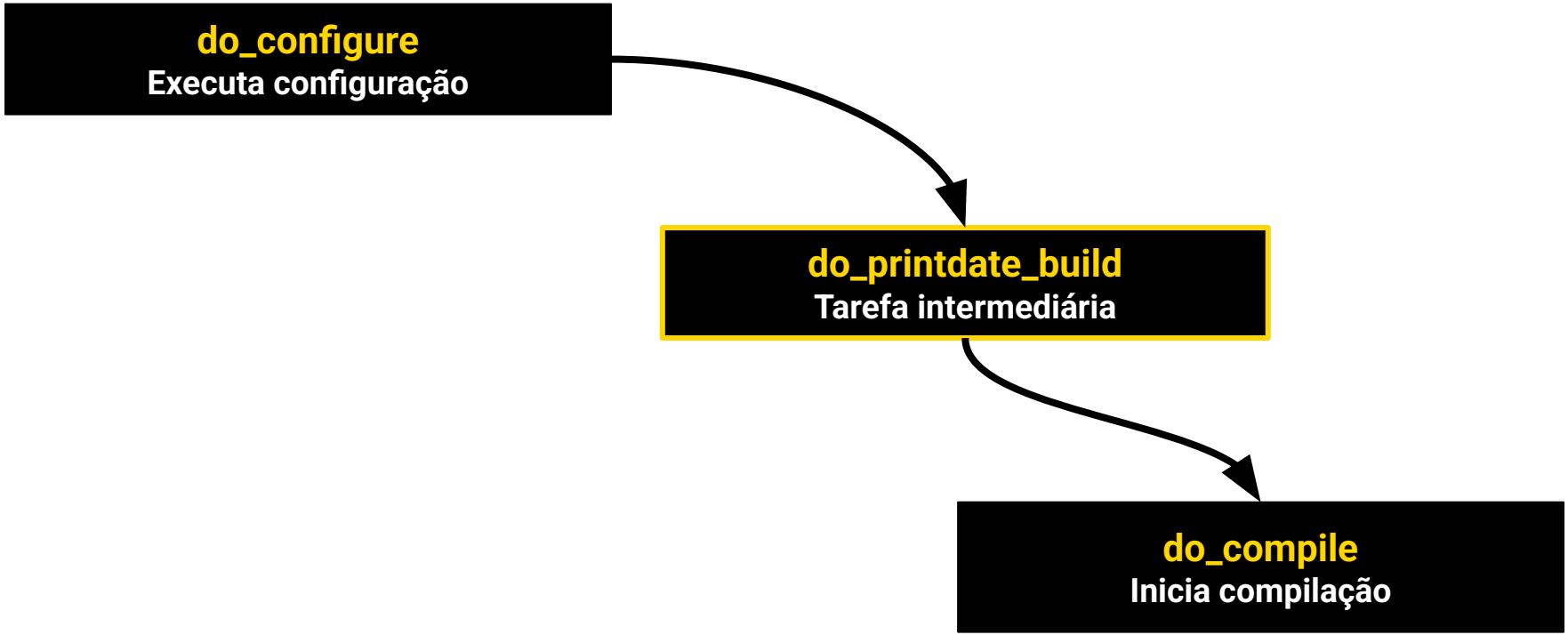
Receita - Adicionando Tarefas

As vezes é necessário executar uma tarefa intermediária durante o fluxo, por exemplo, assinar um binário depois de compilar e antes de iniciar empacotamento.

Utilizando o comando **addtask** pode-se adicionar uma tarefa entre as tarefas.

```
python do_printdate_build () {
        import time
        print time.strftime('%Y%m%d', time.gmtime())
}

addtask printdate_build after do_configure before do_compile
```

Receita - Adicionando Tarefas



Receita - Adicionando Tarefas

As vezes é necessário executar uma tarefa intermediária durante o fluxo, por exemplo, assinar um binário depois de compilar e antes de iniciar empacotamento.

```
python do_generate_hash() {
    import hashlib
    import os
    ...
}

do_fix_extension () {
    ...
}

addtask generate_hash after do_compile before do_package
addtask do_fix_extension after do_configure before do_compile
```

Sintaxe Python

Sintaxe Shell Script

Receita - Removendo Tarefas

Para remover uma tarefa utilize **deltask**:

```
deltask printdate_build
```

Use o **noexec** para desabilitar a tarefa em vez de usar o comando **deltask** para excluí-la:

```
do_printdate_build[noexec] = "1"
```

Laboratório

Prática 06

- Criando a primeira receita
- Receita compilando um código em C
- Receita compilando com Makefile

DEPENDS
RDEPENDS

Configurando dependências para construção e dependências para execução

Receita - DEPENDS e RDEPENDS

DEPENDS

Lista de receitas necessárias para a compilar a receita.

RDEPENDS

Lista de pacotes necessários no TARGET para execução da receita.

HERDAR CLASSES

Herdando classes de arquivos .bbclass para estender ou configurar funcionalidade

inherit

Faz com que a classe ou classes nomeadas sejam herdadas. **ATENÇÃO**, utilizar **INHERIT** a herança é global, utilizado normalmente em .conf.

inherit core-image	Herda recursos para criar a própria receita de imagem

Este arquivo de classe está em **meta/classes/core-image.bbclass**

Receita - Herança

inherit

Classe	Descrição
inherit autotools	A receita possui suporte para processar GNU Autotools
inherit cmake	A receita possui suporte CMake
inherit module	Fornece suporte para a construção de módulos de kernel do Linux fora da árvore
inherit setuptools3	Suporte ao buildsystem SetupTools3 para Python3
inherit npm	Suporte para construção aplicações Node.js
inherit pypi	Fornece suporte para a construção e instalação de pacotes Python a partir do repositório PyPI (Python Package Index)

Receita - Herança

inherit

Classe	Descrição
inherit extrausers	Permite na receita de imagem adicionar usuários para o Sistema - **EXTRA_USERS_PARAMS** [extrauser.bbclass]
inherit adduser	Permite criar usuário/grupo a um pacote, exemplo pulseaudio [adduser.bbclass]
inherit cmake_qt5	Fornece suporte a CMake para projetos em Framework Qt5
inherit qmake5	A receita possui suporte QMake5 para compilar código de Framework Qt5
inherit features_check	Fornece suporte a receita a checagem de recursos de **DISTRO_FEATURES**, **MACHINE_FEATURES** ou **COMBINED_FEATURES**
inherit core-image	Fornece definição básica para uma receita ter definições para receita de imagem

Receita - Herança

inherit

Classe	Descrição
inherit systemd	Permite na receita especificar o arquivo Systemd Unit (app.service) para instalar e poder iniciar o software automaticamente durante a inicialização
inherit uninative	Utilizada para garantir **compatibilidade binária** de artefatos nativos (compilados para a máquina de build) entre **diferentes sistemas host**, como distribuições Linux variadas, ao utilizar um **toolchain nativo pré-compilado(universal)** e isolado; isso evita que ferramentas nativas (como **flex**, **bison**, **pkg-config**, entre outras) sejam compiladas ou apresentem falhas devido a diferenças em bibliotecas do sistema, como a **glibc**, proporcionando builds mais reprodutíveis e estáveis.

Banco de Dados

Site para pesquisa de receitas, machines e camadas

YOCTO PROJECT® COMPATIBLE LAYERS

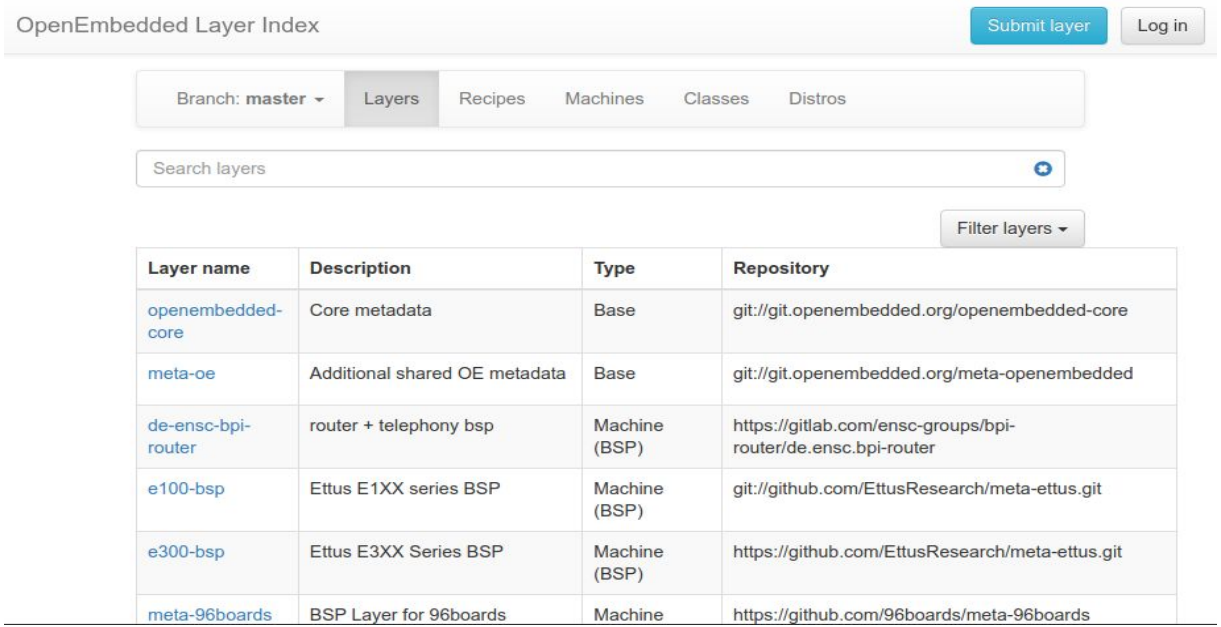
Uma lista mas com um processo de validação mais apurado:

https://www.yoctoproject.org/software-overview/layers/

OpenEmbedded Layer Index

Contém uma lista maior de camadas e um processo com menos validações:

https://layers.openembedded.org



Layer name	Description	Type	Repository
openembedded-core	Core metadata	Base	git://git.openembedded.org/openembedded-core
meta-oe	Additional shared OE metadata	Base	git://git.openembedded.org/meta-openembedded
de-ensc-bpi-router	router + telephony bsp	Machine (BSP)	https://gitlab.com/ensc-groups/bpi-router/de.ensc.bpi-router
e100-bsp	Ettus E1XX series BSP	Machine (BSP)	git://github.com/EttusResearch/meta-ettus.git
e300-bsp	Ettus E3XX Series BSP	Machine (BSP)	https://github.com/EttusResearch/meta-ettus.git
meta-96boards	BSP Layer for 96boards	Machine	https://github.com/96boards/meta-96boards

OpenEmbedded Layer Index

Para submeter uma camada para o **OpenEmbedded Layer Index** pode seguir as orientações de Making Sure Your Layer is Compatible With Yocto Project, e uma ferramenta é fornecida para ajudar nesta análise **yocto-check-layer**.

```
~/yp/poky/build $ yocto-check-layer ../meta-b2open
INFO: Detected layers:
INFO: meta-b2open: LayerType.SOFTWARE, /home/b2open/treinamento/yp/poky/meta-b2open
INFO: Setting up for meta-b2open(LayerType.SOFTWARE), /home/b2open/treinamento/yp/poky/meta-b2open
INFO: Getting initial bitbake variables ...
INFO: Getting initial signatures ...
INFO: Adding layer meta-b2open
INFO: Starting to analyze: meta-b2open
INFO: ----------------------------------------------------------------------
...
INFO: ----------------------------------------------------------------------
INFO: Ran 8 tests in 2272.583s
INFO: OK
INFO:  (skipped=2, unexpected successes=1)
INFO: Summary of results:
INFO: meta-b2open ... PASS
```

Laboratório

Prática 07

- Pesquisar receitas
- Criar receita que utiliza CMake
- Criar receita de um projeto de um repositório Git

COMPATIBLE_MACHINE

Compatibilidade de Machine com Receitas

COMPATIBLE_MACHINE

Expressão regular que define uma ou mais MACHINES de destino com as quais uma receita é compatível. A expressão regular é comparada com **MACHINEOVERRIDES**.

Você pode usar a variável para impedir que receitas sejam criadas para máquinas com as quais as receitas não são compatíveis.

```
COMPATIBLE_MACHINE = "^(mx8|rpi)$"
```

```
COMPATIBLE_MACHINE = "^(qemux86|qemuarm|qemuarm64)$"
```

PACKAGECONFIG

Configurando o build de acordo com recursos

PACKAGECONFIG

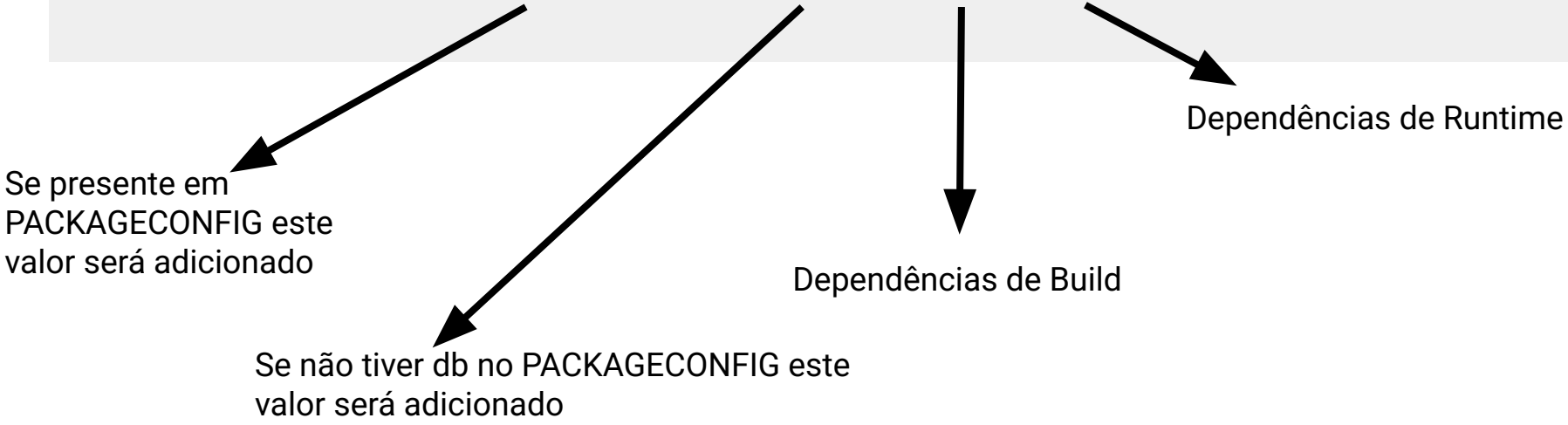
Permite adicionar ou remover flags/parâmetros de compilação adicionando ou removendo valores do **PACKAGECONFIG** ou dinamicamente via Recursos Condicionais.

Altera o comportamento da variável **EXTRA_OECONF**.

PACKAGECONFIG

Essa variável fornece um meio de ativar ou desativar características de uma receita, exemplo:

```
PACKAGECONFIG ??= "ipv6 quiet"
PACKAGECONFIG[ipv6] = "--with-ipv6, --without-f1, ndisc6"
PACKAGECONFIG[quiet] = "--silence,,,"
PACKAGECONFIG[db] = "--enable-sql, --disable-sql, sqlite3, sqlite3"
```



Receita

path: root/meta-oe/recipes-support/nano/nano_4.4.bb

blob: 18121be06e39eb5c320eb68dc353a612b6761dab (plain)

```
DESCRIPTION = "GNU nano (Nano's ANOther editor, or \
Not ANOther editor) is an enhanced clone of the \
Pico text editor."
HOMEPAGE = "http://www.nano-editor.org/"
SECTION = "console/utils"
LICENSE = "GPLv3"
LIC_FILES_CHKSUM = "file://COPYING;md5=f27defe1e96c2e1ecd4e0c9be8967949"

DEPENDS = "ncurses file"
RDEPENDS_${PN} = "ncurses-terminfo-base"

PV_MAJOR = "${@d.getVar('PV').split('.')[0]}"

SRC_URI = "https://nano-editor.org/dist/v${PV_MAJOR}/nano-${PV}.tar.xz"
SRC_URI[md5sum] = "9650dd3eb0adbab6aaa748a6f1398ccb"
SRC_URI[sha256sum] = "2af222e0354848ffaa3af31b5cd0a77917e9cb7742cd073d762f3c32f0f582c7"

inherit autotools gettext pkgconfig

PACKAGECONFIG[tiny] = "--enable-tiny,"
```

Estrutura de uma receita

Ajustando o **PACKAGECONFIG** da receita do gdb:

```
~/yp/poky/build $ cat ../meta-b2open/recipes-devtools/gdb/gdb_14.2.bbappend

PACKAGECONFIG:append = " tui"
```

Ajustando o **PACKAGECONFIG** de uma receita direto do **conf/local.conf**:

PACKAGECONFIG:append:pn-<package-name>

```
PACKAGECONFIG:append:pn-gdb = " tui"
```

Estrutura de uma receita

Na sessão **Dicas Extras e Avançadas** existe um comando para auxiliar em listar todas configurações de **PACKAGECONFIG** e quais flags estão habilitadas em cada receita.

MODIFICANDO RECEITAS

Modificando valores e configurações de receitas via .bbappend

Modificando receitas

Uma receita .bb às vezes necessita de alterações e a maneira correta de realizar é via **.bbappend**.

Modificações que envolvem adicionar novos arquivos, adicionar patches ou modificar/adicionar flags para o processo de compilação.

NUNCA se altera o .bb original de outra camada diretamente nela!

Use **<PN>_%.bbappend** para aplicar uma modificação em todas receitas do <PN>.

Laboratório

Prática 08

- Modificando Receitas
- Pesquisando por bbappends

CLASSES

Reutilizando rotinas entre receitas

Classes

As classes do BitBake são arquivos que contêm variáveis, funções e regras predefinidas que podem ser utilizadas para definir um comportamento ou ação para uma receita ou tarefa. Elas fornecem uma estrutura reutilizável para facilitar o desenvolvimento e a manutenção de receitas no BitBake.

Há implementações em sintaxe **Python** e **Shell Script**.

Possui a extensão .bbclass

Exemplo de Classe é amplamente utilizada é a de empacotamento por exemplo rpm (**package_rpm.bbclass**)

Receitas utilizam **inherit <nome class>** para utilizar as classes

Receitas de imagens

Receitas prontas para gerar uma imagem

Receitas de imagens

Nem toda receita (.bb e .bbappend) descreve uma aplicação em específico, um caso isolado são as receitas de imagens.

Receitas de imagens descreve todos os softwares e algumas configurações a serem instaladas na sua Distribuição Linux, além de configurações e customizações específicas que poderá fazer como nome da imagem, adicionar usuários para o Sistema e afins.

É fornecido uma série de receitas de imagens, que utilizam o prefixo **core-image***, mais detalhes na próxima página.

Receitas de imagens

Receitas	Descrição	Dependência
core-image-minimal	Simples e pequena imagem somente CLI	
core-image-weston	Imagem com suporte Wayland e Weston Compositor	Wayland
core-image-x11	Imagem com suporte X11 para aplicações gráficas	X11
core-image-sato	Imagem com suporte X11 e Desktop Manager Sato	X11

Compilando uma receita de imagem

Para iniciar o processo de compilação de uma receita de software isolada ou imagem, deverá ser utilizado o comando **bitbake**.

Caso seja uma receita de imagem, poderá levar algumas horas e ocupar vários gigabytes de espaço em disco, o primeiro build é o mais demorado até formar o sstate-cache.

```
~/yp/poky/build $ bitbake core-image-minimal
```

Crie sua própria receita de imagem

Para criar sua receita de imagem o requisito inicial é herdar **core-image** e ir adicionando as necessidades do seu projeto e customizações.

```
SUMMARY = "Simples de receita de imagem console referencia para IoT"

IMAGE_FEATURES += "splash package-management"

LICENSE = "MIT"

IMAGE_INSTALL:append = " htop sqlite3 libgpiod libgpiod-tools"

inherit core-image
```

Adicionando Usuários

Uma classe chamada **extrausers** oferece suporte para adicionar ou alterar usuários na geração da imagem.

```
inherit extrausers

EXTRA_USERS_PARAMS = " \
   useradd -p 'suport3' suporte; \
   "
```

Referência: https://docs.yoctoproject.org/3.1.31/ref-manual/ref-classes.html#extrausers-bbclass

Adicionando Usuários

Uma classe chamada **extrausers** oferece suporte para adicionar ou alterar usuários na geração da imagem.

printf "%q" $(mkpasswd -m sha256crypt suporte3)

```
inherit extrausers

PASSWD = "\$5\$Pa7JqDV5A8l9/pMP\$<LONG_HASH>"

EXTRA_USERS_PARAMS = "\
    useradd -p '${PASSWD}' suporte; \
    "
```

Referência: https://docs.yoctoproject.org/4.0.16/ref-manual/classes.html#extrausers

Laboratório

Prática 09

- Criando sua receita de imagem
- Adicionando usuários

Camadas de BSP

Propósito em especificar e suportar o hardware

MACHINE

Especificação de hardware

MACHINE

A configuração de hardware segue a topologia:

```
meta-<camada>/conf/machine
```

Exemplos:

```
meta-toradex-nxp/conf/machine/colibri-imx8x.conf
meta-raspberrypi/conf/machine/raspberrypi4.conf
meta-ti/meta-ti-bsp/conf/machine/beaglebone.conf
```

O nome do arquivo refere-se ao nome do **MACHINE**, exemplo:

```
MACHINE ?= "raspberrypi4"
```

MACHINE

Criar um novo arquivo de configuração de hardware requer algumas conhecimento breves como a Arquitetura da CPU para o correto suporte.

A variável **MACHINE_FEATURES** especifica uma lista de valores dos recursos do hardware como: **wifi bluetooth touchscreen serial**

A variável **SERIAL_CONSOLE** recebe o valor da porta serial utilizada como interface serial com o usuário(getty), exemplo: **115200;ttymxc0**

A variável **KERNEL_DEVICETREE** lista dos device-trees que deverá ser instalado.

A variável **MACHINEOVERRIDES** permite definir overrides para a placa alvo.

MACHINE

Duas variáveis importantes são as que definem a receita do Bootloader e Kernel e suas respectivas versões a serem utilizadas, exemplo:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-toradex"
PREFERRED_VERSION_linux-toradex = "5.15%"


PREFERRED_PROVIDER_virtual/bootloader = "u-boot-toradex"
PREFERRED_VERSION_u-boot-toradex = "2022.04%"
```

Bootloader

Modificando valores e configurações de receitas via .bbappend

BOOTLOADER

A receita do bootloader normalmente fica em:

```
meta-<camada>/recipes-bsp/<bootloader-name>
```

Exemplos:

```
meta-toradex-nxp/recipes-bsp/u-boot/u-boot-toradex_2022.04.bb

meta-ti/meta-ti-bsp/recipes-bsp/u-boot/u-boot-ti-staging_2023.04.bb
```

BOOTLOADER

Algumas variáveis específicas do Bootloader, exemplo UBoot:

Variável	Descrição
UBOOT_SUFFIX	Sufixo da extensão, padrão .bin ou .img
UBOOT_MACHINE	O defconfig utilizada para configuração do Bootloader para o Hardware
UBOOT_ENTRYPOINT	Endereço de onde o Bootloader será executado
UBOOT_LOADADDRESS	Endereço para onde o Bootloader será carregado

Kernel

Receitas de kernel, fragmentos e variáveis específicas

KERNEL

A receita do Kernel normalmente fica em:

```
meta-<camada>/recipes-kernel/linux
```

Exemplos:

```
meta-toradex-nxp/recipes-kernel/linux/linux-toradex_5.15-2.1.x.bb

meta-ti/meta-ti-bsp/recipes-kernel/linux/linux-ti-staging_5.10.bb
```

KERNEL

Algumas variáveis específicas o Kernel:

Variável	Descrição
KERNEL_DEVICETREE	Lista de arquivos .dtb para instalar no Target
KERNEL_MODULE_LOAD	Lista de módulos para carregar durante a inicialização
KBUILD_DEFCONFIG	Arquivo defconfig para configuração do Kernel do Target

Camadas de Distro

Foco em descrever e configurar Distribuições

DISTRO

A configuração de DISTRO segue a topologia:

```
meta-<camada>/conf/distro
```

Exemplos:

```
meta-toradex-distro/conf/distro/tdx-xwayland.conf
meta-yocto/meta-poky/conf/distro/poky.conf
meta-webos-ports/meta-luneos/conf/distro/luneos.conf
```

O nome do arquivo refere-se ao nome do **DISTRO**, exemplo:

```
DISTRO ?= "tdx-xwayland"
```

DISTRO

Algumas variáveis específicas para DISTRO:

Variável	Descrição
DISTRO	O nome abreviado/curto da Distribuição
DISTRO_NAME	Nome da Distribuição (informação a ser inserir na imagem)
DISTRO_VERSION	Versão da Distribuição
DISTRO_FEATURES	Recursos que a Distribuição possui como x11, wayland, alsa, pulseaudio, ...
SDK_VENDOR	Especifica o nome do SDK
INIT_MANAGER	Sistema de Inicialização, exemplo: systemd

DISTRO

Um exemplo de uma **DISTRO** em **meta-<camada>/conf/distro/b2os.conf**.

```
require conf/distro/poky.conf

DISTRO = "b2os"
DISTRO_NAME = "B2Open Embedded Linux OS"
DISTRO_VERSION = "4.0"

SDK_VENDOR = "-b2ossdk"

TARGET_VENDOR = "-b2os"

DISTRO_FEATURES:append = " x11 wayland"
DISTRO_FEATURES:remove = "3g wifi bluetooth"

INIT_MANAGER ?= "systemd"
```

Laboratório

Prática 10

- Criando uma DISTRO

Receita de Kernel

Customizando o Kernel

KERNEL

Possui o mesmo formato das receitas .bb mas possui uma particularidade que é herdar a classe **kernel.bbclass**.

Estendendo a receita de kernel para carregar e aplicar um defconfig e fragmentos.

Até mesmo o bitbake possui algumas tarefas extras específicas a receita de kernel.

KERNEL

Podemos criar arquivos .cfg que são fragmentos de configurações habilitando ou desabilitando recursos.

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI  += " \
                file://defconfig \
                file://touchscreen.cfg \
              "
```

```
~/yp/poky/build $ cat ../meta-b2open/recipes-kernel/linux/files/touchscreen.cfg

CONFIG_TOUCHSCREEN_GOODIX=m
```

KERNEL

Aplicar patches no kernel é uma tarefa comum, assim como no Kernel os patches podem ser aplicados em qualquer receita.

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI  += " \
                  file://defconfig \
                  file://touchscreen.cfg \
                  file://fix-timeout-usdhc.patch \
                "
```

Laboratório

Prática 11

- Customizando Kernel Linux
- Criando Patch
- Criando receita modificação Kernel Linux

Receitas

Debugging

Debugging Bitbake

Obtendo configuração das variáveis:

```
~/yp/poky/build $ bitbake -e | grep ^DISTRO_FEATURES
```

Obtendo configuração das variáveis de uma receita:

```
~/yp/poky/build $ bitbake -e busybox | grep ^PACKAGES
```

Debugging Bitbake

Listando as tarefas disponíveis da receita:

```
~/yp/poky/build $ bitbake -c listtasks busybox
```

Executando uma tarefa em específica para a receita, exemplo **do_compile**:

```
~/yp/poky/build $ bitbake -c compile -f busybox
```

Debugging Bitbake

Habilitando Debug Information no bitbake:

```
~/yp/poky/build $ bitbake -DDD busybox
```

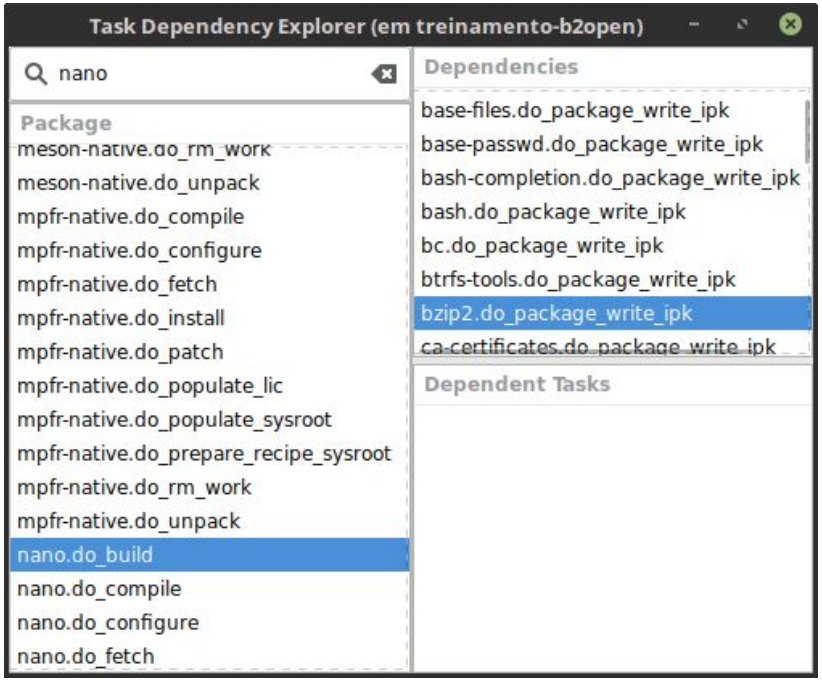
Executando bitbake com modo verbose:

```
~/yp/poky/build $ bitbake -v busybox
```

Debugging Bitbake

Analisando dependências de uma receita:

```
~/yp/poky/build $ bitbake -g -u taskexp nano
```



Toolchain/SDK

Toolchain/SDK

A Poky permite gerar dois tipos de SDK, este que irá gerar todo toolchain para ser instalado no HOST(computador de desenvolvimento) para realizar compilação-cruzada para a arquitetura do TARGET, com todas diretivas corretamente configuradas e exportadas.

As opções são:

- SDK Genérico
- SDK Completo
- Extensible SDK

Toolchain/SDK - SDK Genérico

SDK Genérico - Este método irá construir um conjunto de ferramentas que corresponda à sua plataforma do TARGET e um conjunto básico do sysroot sem as dependências e todos recursos do sysroot final.

Este tipo de SDK é utilizado para compilar softwares como Bootloader, Kernel Linux e aplicações que não precisem de um sysroot.

Toolchain/SDK - SDK Genérico

Para gerar o SDK Genérico:

```
b2open@yp-b2open:~/yp/build-toradex $ bitbake meta-toolchain
```

Toolchain/SDK - SDK Genérico

O SDK Genérico será gerado em **tmp/deploy/sdk**.

```
b2open@yp-b2open:~/yp/build-toradex $ ls tmp/deploy/sdk

tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0.host.manifest
tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0-host.spdx.tar.zst
tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0.sh
tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0.target.manifest
tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0-target.spdx.tar.zst
tdx-xwayland-glibc-x86_64-meta-toolchain-armv8a-colibri-imx8x-toolchain-7.3.0.testdata.json
```

Toolchain/SDK - SDK Completo

SDK Completo - Este método irá construir além do SDK Genérico, irá incluir o sysroot idêntico ao gerado para a plataforma do TARGET, inclui as ferramentas e bibliotecas/cabeçalhos adicionados na imagem.

Este tipo de SDK é utilizado para compilar todos os softwares que possuem alguma das dependências adicionadas na imagem.

Toolchain/SDK - SDK Completo

Para gerar o SDK Completo:

```
b2open@yp-b2open:~/yp/build-toradex $ bitbake core-image-minimal -c populate_sdk
```

Toolchain/SDK - SDK Completo

O SDK Completo será gerado em **tmp/deploy/sdk**, mas o script de instalação é o arquivo .sh.

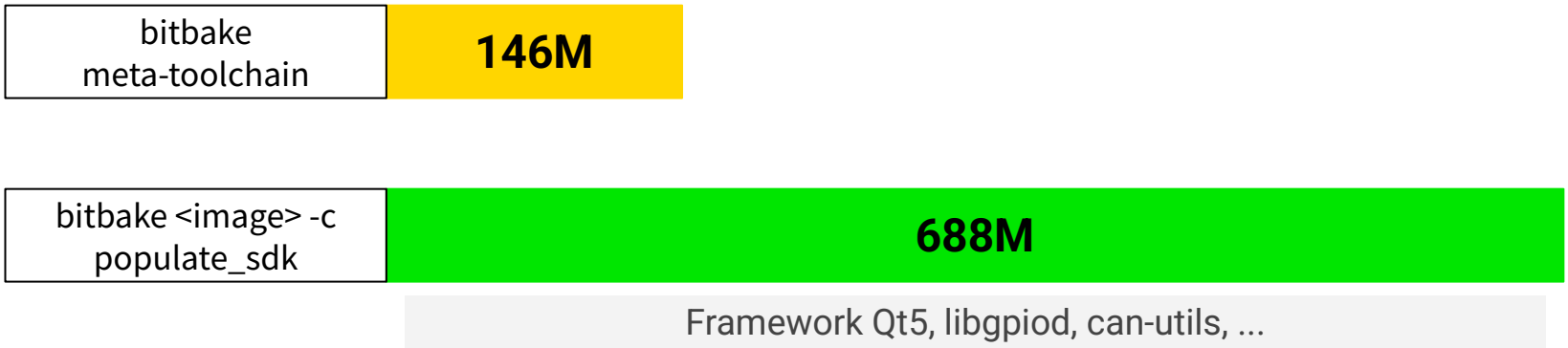
```
b2open@yp-b2open:~/yp/build-toradex $ ls tmp/deploy/sdk

tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0.host.manifest
tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0-host.spdx.tar.zst
tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0.sh
tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0.target.manifest
tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0-target.spdx.tar.zst
tdx-xwayland-glibc-x86_64-core-image-minimal-armv8a-colibri-imx8x-toolchain-7.3.0.testdata.json
```

Toolchain/SDK

Comparativo do script gerado:

Toolchain/SDK

Comparativo do script gerado:



Toolchain/SDK - SDK Completo

Extensible SDK - Este método irá construir além do SDK Completo, inclui ferramentas adicionais para administrar receitas como **devtool**.

Toolchain/SDK - SDK Qt5

O mesmo vale para Framework Qt5.

```
b2open@yp-b2open:~/yp/build-toradex $ bitbake meta-toolchain-qt5
```

```
b2open@yp-b2open:~/yp/build-toradex $ bitbake core-image-minimal -c populate_sdk
```

Toolchain/SDK - Instalação

Ambos os SDK gerados ao executar será instalado no diretório padrão **/opt/poky/<POKY_VERSION>**.

Ao executar o script **.sh** de instalação, poderá especificar outro destino para instalação.

Um arquivo chamado environment-setup-cortexa7hf-neon-poky-linux-gnueabi será gerado, onde deverá ser exportado via comando source para definir todas variáveis ambiente antes de abrir a IDE.

```
$ source /opt/sdk/3.1.7/environment-setup-cortexa7hf-neon-poky-linux-gnueabi

$ code
```

Laboratório

Prática 12

- Instalando Toolchain
- Realizando compilação cruzada

LICENÇAS

Gerenciando e restringindo licenças

LICENÇAS

As licenças dos pacotes de software desempenham um papel importante. O bitbake oferece várias funcionalidades para gerenciar as licenças dos pacotes incluídos em uma distribuição. Algumas variáveis importantes referentes a LICENÇAS:

Variável	Descrição
INCOMPATIBLE_LICENSE	Especificas licenças incompatíveis com o projeto e o bitbake pode acusar erro caso encontre
LICENSE_FLAGS	Variável utilizada em cada pacote para especificar a licença do software
COPY_LIC_DIRS	Copia todas as licenças dos pacotes utilizados para o diretórios **/usr/share/common-licenses** dentro do TARGET
COPY_LIC_MANIFEST	Gera em um único arquivo todas as licenças utilizadas e cópia para o TARGET

LICENÇAS

As licenças dos pacotes de software desempenham um papel importante. O bitbake oferece várias funcionalidades para gerenciar as licenças dos pacotes incluídos em uma distribuição. Algumas variáveis importantes referentes a LICENÇAS:

Variável	Descrição
INHERIT += "archiver" ARCHIVER_MODE[src] = "original"	Ainda no contexto de compliance, esta configuração gera a cópia do código-fonte, patches e licenças no final da construção em **<build>/tmp/deploy/sources**
LICENSE_FLAGS_ACCEPTED	Permite especificar uma licença restrita ao projeto, configurando na Distribuição ou em <build/conf/local.conf: LICENSE_FLAGS_ACCEPTED = "commercial" LICENSE_FLAGS_ACCEPTED = "commercial_gstreamer-1.0-plugins-ugly"

Yocto Project

Avançado e Extras

DEVTOOL

Integração e desenvolvimento de receitas

DEVTOOL

Uma ferramentas que pode ser utilizada pelo ambiente configurado para build ou via SDK gerado

Permite especificar um repositório GIT ou diretório local e um nome e irá criar a receita .bb para este projeto

Além de outros comandos como **add**(cria receita), **build**(processa receita), **edit-recipe**(edita receita), **deploy-target**(enviar aplicação para TARGET), **undeploy-target**(desinstalar aplicação do TARGET, **finish**(cópia receita para uma camada) e **reset**(permite limpar receita e todo conteúdo do workspace).

Laboratório

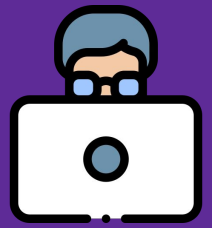
Prática 13

- Prática devtool

Bônus

Avançado e Extras

- Configurando Hostname
- Rotinas após criar RootFS
- Auto-Revision GIT
- Python Code Expansion
- Shell Script Expansion



Gerenciando Camadas

Distribuir os fontes de uma Distribuição

Gerenciando Camadas

repo (Google) - https://source.android.com/docs/setup/download#repo

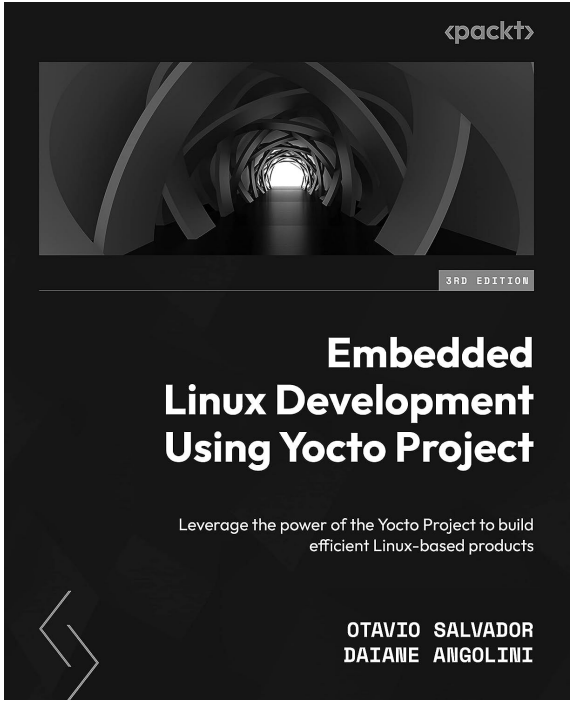
- Toradex - Build a Reference Image with Yocto Project

kas (Siemens) - https://github.com/siemens/kas

whisk (Garmin) - https://github.com/garmin/whisk

Yocto Buddy (Agilent) - https://github.com/Agilent/yp

Livros para estudo



Obrigado!

Cleiton Bueno - cleiton.bueno@b2open.com