SOMMAIRE

SOMMAIRE

Fondamentaux de Yocto

Introduction aux composants clés de Yocto.

Yocto est un projet open-source qui permet de créer des systèmes Linux personnalisés pour des appareils embarqués.

1. **Poky** : C'est la distribution de référence de Yocto. Poky est un modèle de départ qui inclut le script `oe-init-build-env`, l'ensemble de métadonnées de base, un ensemble de configurations prédéfinies, ainsi que BitBake, l'outil de build central de Yocto.
2. **BitBake** : C'est l'outil de build principal utilisé par Yocto. Il sert à exécuter les tâches telles que le téléchargement des sources, leur configuration, la compilation et l'assemblage des paquets logiciels. BitBake utilise des recettes et des classes pour gérer les dépendances et orchestrer le processus de construction.
3. **Recettes** : Les recettes sont des fichiers qui fournissent les instructions nécessaires à BitBake pour construire un paquet spécifique. Elles contiennent des métadonnées sur les sources du paquet, les dépendances, les instructions de compilation et d'installation, etc.
4. **Layers** : Les layers sont des collections de recettes et de classes qui permettent de modulariser et de réutiliser les configurations. Par exemple, un layer peut contenir tout le nécessaire pour supporter un type particulier de matériel ou une pile logicielle spécifique.

Fondamentaux de Yocto

Introduction aux composants clés de Yocto.

5. **Meta-Layers** : Ce sont des layers qui contiennent des informations sur d'autres layers. Par exemple, `meta-oe` est un layer qui contient des recettes supplémentaires pour des paquets couramment utilisés dans des systèmes embarqués.
6. **Toolchain** : Yocto permet de générer des toolchains croisées, qui sont des ensembles d'outils de compilation permettant de construire du code pour une architecture matérielle cible à partir d'une architecture hôte différente.
7. **Image Builder** : C'est un outil utilisé pour créer les images système finales à partir des paquets compilés. Cela inclut les systèmes de fichiers pour divers formats de périphériques et médias de stockage.
8. **SDK** (Software Development Kit) : Yocto peut générer un SDK spécifique au projet, permettant aux développeurs de créer des applications pour le système embarqué spécifique qu'ils ont construit.
9. **OE-Core** : C'est le cœur des données de base de OpenEmbedded, une partie importante de l'écosystème Yocto qui fournit les éléments essentiels nécessaires à la création d'un système d'exploitation Linux.

Fondamentaux de Yocto

Aperçu de l'architecture et des outils de Yocto

L'architecture de Yocto est conçue pour être modulaire et extensible, ce qui permet de créer des systèmes embarqués personnalisés en utilisant une série de composants interconnectés.

1. **BitBake** :
   - Cœur du système Yocto, c'est un moteur de tâches qui interprète les recettes et orchestre le processus de construction.
2. **Métadonnées** :
   - **Recettes** (`*.bb`) : Fichiers qui contiennent les instructions pour construire un paquet spécifique, y compris les sources, dépendances, et étapes de compilation.
   - **Classes** (`*.bbclass`) : Fichiers réutilisables qui fournissent des fonctions partagées entre les recettes.
   - **Configurations** (`*.conf`) : Fichiers de configuration pour les builds, y compris la configuration de la machine cible, les préférences de l'utilisateur, et les variables globales.

Fondamentaux de Yocto

Aperçu de l'architecture et des outils de Yocto

3. **Layers** :
    - Collections organisées de recettes, classes, et configurations spécifiques à un domaine (par exemple, support matériel, interfaces utilisateur graphiques, etc.).
4. **OpenEmbedded Core (OE-Core)** :
    - Ensemble de recettes, classes, et configurations de base qui fournissent les fonctionnalités essentielles pour un système Linux minimal.

Fondamentaux de Yocto

Aperçu de l'architecture et des outils de Yocto

- `build/` :
  - `conf/` : Contient les fichiers de configuration principaux comme `local.conf` et `bblayers.conf`.
  - `tmp/` : Le dossier de travail principal pour les builds, subdivisé en plusieurs sous-dossiers :
    - `deploy/` : Où les images finales et les paquets sont stockés après la construction.
    - `work/` : Contient les répertoires de travail pour chaque recette construite, organisés par architecture et nom de paquet.
    - `cache/` : Stocke les données de cache qui aident à accélérer le processus de build en réutilisant les résultats antérieurs.
    - `sysroots/` : Contient les systèmes racine pour chaque architecture cible et pour l'hôte, utilisés pour la compilation croisée.
    - `stamps/` : Dossiers contenant des "timestamps" indiquant quand les tâches spécifiques ont été complétées, utilisés pour gérer les dépendances entre tâches.

Fondamentaux de Yocto

Aperçu de l'architecture et des outils de Yocto

- `sources/` :
  - `meta/` : Layer de base contenant des recettes essentielles et des classes.
  - `meta-yocto/` : Contient des recettes spécifiques au projet Yocto, telles que les recettes pour l'image de démarrage de Poky.
  - `meta-yocto-bsp/` : Layer pour les BSPs.
  - `meta-<nom>` : D'autres layers pour des fonctionnalités ou des plateformes spécifiques.
  - `poky/` : Parfois présent, surtout si Poky est utilisé comme référence. Poky inclut BitBake, la documentation, les scripts de démarrage, et les métadonnées de base.
- `downloads/` : Un dossier partagé pour stocker les sources téléchargées de tous les paquets.
- `sstate-cache/` : Stocke les composants pré-compilés pour accélérer les builds répétitifs.
- `cache/` : Parfois utilisé pour conserver des données de configuration et de parsing de BitBake pour améliorer les performances des builds successifs.

Fondamentaux de Yocto

Aperçu de l'architecture et des outils de Yocto

- **`scripts/`** :
  - Contient des scripts utilisés pour des tâches telles que l'initialisation de l'environnement de build, l'analyse des logs, et la gestion des layers.
- **`oe-init-build-env`** : Script pour initialiser l'environnement de build. Il configure les variables d'environnement nécessaires pour utiliser BitBake et prépare le dossier `build/`.
- **`bitbake-layers`** : Utilitaire pour aider à gérer les layers dans `bblayers.conf`.
- **`local.conf`** et **`bblayers.conf`** : Comme déjà mentionné, ce sont des fichiers clés pour la configuration du processus de build, spécifiant les layers à utiliser, les paramètres de la machine cible, les options de compilation, etc.

Travaux Pratiques

1. **Installer les paquets nécessaires** :

- Pour Ubuntu :

```
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
```

2. **Cloner le dépôt Poky** :

- Clonez le dépôt Poky depuis GitHub :

```
git clone git://git.yoctoproject.org/poky
cd poky
```

3. **Initialiser l'environnement** :

- Sourcez le script d'initialisation pour configurer l'environnement :

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

Travaux Pratiques

4. **Cloner les layers nécessaires** :

- Vous devez ajouter le BSP pour Raspberry Pi. Clonez `meta-raspberrypi` dans le dossier `sources/` :

```
git clone git://git.yoctoproject.org/meta-raspberrypi sources/meta-raspberrypi
```

5. **Configurer bblayers.conf** :

- Éditez le fichier `conf/bblayers.conf` pour inclure `meta-raspberrypi` :

```
BBLAYERS ?= " \
   /chemin/vers/poky/meta \
   /chemin/vers/poky/meta-poky \
   /chemin/vers/poky/meta-yocto-bsp \
   /chemin/vers/poky/meta-raspberrypi \
"
```

Travaux Pratiques

6. **Modifier local.conf** :

- Éditez le fichier `conf/local.conf` :
- Définissez la machine cible :

```
MACHINE ??= "raspberrypi3"
```

7. **Construire l'image** :

- Construisez une image simple, telle que `core-image-minimal` :

```
bitbake core-image-minimal
```

Création et gestion de layers personnalisés.

1. **Initialiser l'Environnement de Construction** :

```
source oe-init-build-env /home/yocto/poky/build
```

2. **Créer un Nouveau Layer** :

```
bitbake-layers create-layer meta-custom
```

3. **Ajouter le Layer à bblayers.conf** :

- Ajoutez le nouveau layer à `bblayers.conf`.

```
bitbake-layers add-layer meta-custom
```

4. **Activer/Désactiver un Layer** :
   - Pour ajouter un layer :

```
bitbake-layers add-layer <layer-path>
```

   - Pour supprimer un layer :

```
bitbake-layers remove-layer <layer-path>
```

Création et gestion de layers personnalisés - Ajout de Packages.

1. **Modifier la Recette d'Image** :
   - Ouvrez la recette de l'image que vous souhaitez modifier, par exemple `core-image-minimal.bb`.

```
nano meta-custom/recipes-core/images/core-image-custom.bb
```

2. **Ajouter les Packages** :
   - Ajoutez les packages que vous souhaitez inclure dans l'image.

```
require recipes-core/images/core-image-minimal.bb
IMAGE_INSTALL += "package1 package2"
```

3. **Supprimer les Packages** :
   - Utilisez `IMAGE_INSTALL_remove` pour supprimer les packages.

```
require recipes-core/images/core-image-minimal.bb
IMAGE_INSTALL_remove = "package-to-remove"
```

Types de Fichiers BitBake

1. **.bb (BitBake Recipe Files)**
   - **Description** : Les fichiers `.bb` contiennent des recettes BitBake. Une recette décrit comment obtenir, configurer, compiler et installer un paquet logiciel.
   - **Utilisation** : Utilisé pour créer des paquets spécifiques.
2. **.bbappend (BitBake Append Files)**
   - **Description** : Les fichiers `.bbappend` permettent d'ajouter ou de modifier des informations dans des recettes `.bb` existantes sans les dupliquer.
   - **Utilisation** : Utilisé pour ajouter des patches, modifier des variables, ou ajouter des étapes à une recette existante.
3. **.bbclass (BitBake Class Files)**
   - **Description** : Les fichiers `.bbclass` contiennent des classes BitBake. Une classe est un ensemble de fonctions et de variables qui peuvent être réutilisées dans plusieurs recettes.
   - **Utilisation** : Utilisé pour partager des comportements communs entre plusieurs recettes.

Types de Fichiers BitBake

4. **.inc (Include Files)**
   - **Description** : Les fichiers `.inc` sont des fichiers d'inclusion qui permettent de partager des fragments de recettes entre plusieurs recettes `.bb`.
   - **Utilisation** : Utilisé pour éviter la duplication de code en incluant des variables et des fonctions communes.
5. **.conf (Configuration Files)**
   - **Description** : Les fichiers `.conf` contiennent des configurations globales ou spécifiques à une build.
   - **Utilisation** : Utilisé pour configurer l'environnement de build, les machines cibles, et d'autres paramètres globaux.
   - **Exemple** : `local.conf`, `bblayers.conf`

Quand Utiliser Chaque Type de Fichier

- **.bb** : Utilisé pour définir des recettes individuelles pour des paquets logiciels spécifiques.
- **.bbappend** : Utilisé pour modifier ou étendre des recettes `.bb` existantes, particulièrement utile pour appliquer des patches ou des modifications spécifiques sans dupliquer toute la recette.
- **.bbclass** : Utilisé pour encapsuler des comportements réutilisables qui peuvent être inclus dans plusieurs recettes.
- **.inc** : Utilisé pour partager des fragments de code entre plusieurs recettes, évitant ainsi la duplication de code.
- **.conf** : Utilisé pour configurer l'environnement de build, incluant des configurations globales, des machines cibles, et des couches (layers).

Cycle de vie d'un fichier .bb

1. fetch (do_fetch) : Télécharger les sources.
2. unpack (do_unpack) : Décompresser les sources.
3. patch (do_patch) : Appliquer des patches.
4. configure (do_configure) : Configurer les sources pour la compilation.
5. compile (do_compile) : Compiler les sources.
6. install (do_install) : Installer les fichiers compilés dans le répertoire de destination.
7. package (do_package) : Créer les packages binaires.

TP 1

Objectif : Créer un custom layer nommé `meta-networking` et y ajouter une recette pour une application réseau `net-app` en utilisant toutes les étapes du cycle de vie d'une recette Yocto (fetch, unpack, patch, configure, compile, install, package).
Description de l'Application : Vous allez développer une application réseau simple en C appelée `net-app` qui écoute sur le port 8080 et renvoie un message prédéfini aux clients qui se connectent. Cette application sera intégrée dans un environnement Yocto. Vous devrez créer un custom layer, développer une recette Yocto pour cette application, et suivre toutes les étapes du cycle de vie d'une recette Yocto.

Partie 1 : Créer et Intégrer l'Application Réseau

1. **Préparer les Fichiers Sources** :
2. **Créer la Structure du Custom Layer** :
3. **Ajouter la Recette de l'Application** :
4. **Ajouter le Custom Layer au Build** :
5. **Construire l'Image** :

TP 1

Partie 2 : Appliquer un Patch à l'Application

1. **Créer un Patch pour `net-app.c`** :
2. **Ajouter le Patch à la Recette** :
3. **Reconstruire l'Image** :

TP 2

Vous allez modifier le noyau Linux utilisé par Yocto pour ajouter un pilote de périphérique. Pour cela, vous devrez créer un custom layer, développer une recette Yocto pour appliquer un patch au noyau Linux.

Introduction à l'Application Development Toolkit (ADT) de Yocto

- L'Application Development Toolkit (ADT) de Yocto est un ensemble d'outils et de méthodologies conçu pour aider les développeurs à créer, développer, tester et maintenir des applications pour des systèmes embarqués utilisant le Yocto Project. Le Yocto Project lui-même est une initiative open source qui fournit des modèles, des outils, et des méthodes pour créer des systèmes d'exploitation Linux personnalisés pour des dispositifs embarqués.
- L'objectif principal de l'ADT est de simplifier le développement d'applications pour les plateformes supportées par Yocto. Il est particulièrement utile pour les développeurs qui ont besoin de créer des applications spécifiques à une plateforme, optimisées pour un certain type de matériel embarqué.

Introduction à l'Application Development Toolkit (ADT) de Yocto

Composants de l'ADT de Yocto

L'ADT comprend plusieurs composants clés qui facilitent le développement d'applications embarquées :

1. **Cross-Compiler Toolchains** :
   - Des chaînes d'outils de compilation croisée qui permettent aux développeurs de compiler du code sur une architecture hôte (par exemple, x86_64) qui sera exécuté sur une architecture cible différente (par exemple, ARM).
2. **Emulateurs (QEMU)** :
   - Des émulateurs intégrés tels que QEMU pour tester les applications et le système d'exploitation dans un environnement simulé avant le déploiement sur le matériel réel.
3. **Environnements de développement (IDE plugins)** :
   - Des plugins pour des environnements de développement intégrés populaires comme Eclipse, permettant une intégration plus profonde et une meilleure expérience utilisateur pour le développement d'applications.

Introduction à l'Application Development Toolkit (ADT) de Yocto

4. **SDK (Software Development Kit)** :
   - Des kits de développement logiciel qui contiennent tous les outils nécessaires pour construire des applications, y compris des bibliothèques, des en-têtes et des outils de ligne de commande qui correspondent à l'environnement de la cible.
5. **Outils de débogage et de profilage** :
   - Des outils pour le débogage et le profilage d'applications qui aident les développeurs à optimiser leurs applications pour des performances et une efficacité maximales.

Introduction à l'Application Development Toolkit (ADT) de Yocto

Utilisation de l'ADT

L'utilisation de l'ADT commence généralement par la génération d'un SDK personnalisé pour un projet spécifique.

1. **Génération du SDK** :
   - Utilisez Yocto pour créer un SDK qui correspond à l'architecture cible et qui comprend les bibliothèques et les outils nécessaires.
2. **Configuration de l'environnement de développement** :
   - Installez le SDK généré sur votre système de développement et configurez l'environnement pour utiliser les chaînes d'outils et les bibliothèques incluses.
3. **Développement d'applications** :
   - Développez des applications en utilisant les outils fournis par le SDK. Testez les applications localement sur l'émulateur QEMU ou déboguez-les à l'aide des outils disponibles.
4. **Déploiement et tests** :
   - Déployez les applications sur le matériel cible et effectuez des tests finaux pour assurer leur bon fonctionnement dans l'environnement réel.

Introduction à l'Application Development Toolkit (ADT) de Yocto

Avantages de l'ADT de Yocto

- **Uniformité et standardisation** : L'ADT offre un ensemble standard d'outils et de méthodes pour le développement d'applications, ce qui facilite la collaboration et la gestion des projets au sein des équipes.
- **Optimisation** : Les applications peuvent être optimisées pour une utilisation efficace des ressources du matériel cible.
- **Flexibilité** : Les développeurs peuvent facilement adapter et configurer l'environnement de développement pour répondre aux besoins spécifiques de leur projet.

Le SDK (Software Development Kit)

Le SDK (Software Development Kit) généré par Yocto est un ensemble complet d'outils et de ressources conçu pour aider les développeurs à compiler et tester des applications pour des systèmes embarqués spécifiques.

1. Compilateurs Cross: Le SDK inclut des compilateurs cross GCC qui permettent de compiler du code pour l'architecture de la plate-forme cible depuis une machine hôte (généralement x86_64). Cela comprend:

- **gcc** : Le compilateur GNU pour C et C++.
- **g++** : Le compilateur GNU pour C++.
- **gfortran** : Le compilateur pour Fortran, si inclus.

2. Bibliothèques Standard et de Développement: Ces bibliothèques sont nécessaires pour le développement et l'exécution d'applications. Elles comprennent les versions cross-compilées des bibliothèques standard du C et d'autres bibliothèques communes nécessaires pour le développement d'applications, telles que:

- **libc** (la bibliothèque standard du C),
- **libstdc++** (la bibliothèque standard C++),
- **libm** (la bibliothèque de mathématiques),
- et d'autres bibliothèques pertinentes selon la configuration du build Yocto (comme OpenSSL, zlib, etc.).

3. Outils de Débogage

- **GDB** (GNU Debugger) pour le débogage des applications. Le SDK inclut souvent une version cross-compilée de GDB qui peut être utilisée pour déboguer des applications en cours d'exécution sur le matériel cible.

Le SDK (Software Development Kit)

4. Outils de Profilage: Des outils tels que `gprof` ou `oprofile` peuvent également être inclus pour aider au profilage des applications et à l'analyse des performances.

5. Bibliothèques de Développement et En-têtes: Le SDK inclut les en-têtes et bibliothèques nécessaires pour développer des applications utilisant des fonctionnalités spécifiques du matériel cible ou des bibliothèques système supplémentaires.

*__6. Utilitaires de la Ligne de Commande__: Des outils tels que `make`, `pkg-config`, et d'autres utilitaires utiles pour le développement de logiciels.

7. Scripts d'Environnement: Des scripts qui configurent l'environnement de développement en définissant des variables d'environnement nécessaires telles que `CC`, `CXX`, `LD`, `ARCH`, `CROSS_COMPILE`, et d'autres qui facilitent le développement cross.

8. Outils de Construction

- **bitbake** : Un outil pour automatiser la construction et la gestion des packages, bien que son utilisation directe ne soit pas toujours incluse dans le SDK.
- **Sysroots** : Environnements isolés qui contiennent des copies des bibliothèques système, des en-têtes, et d'autres fichiers essentiels à la compilation des applications.

Introduction aux Systèmes d'Exploitation Temps Réel (RTOS)

- Un Système d'Exploitation Temps Réel (RTOS) est conçu pour gérer les ressources matérielles et logicielles d'un système informatique en garantissant que certaines tâches critiques sont exécutées dans des délais prévisibles. Contrairement aux systèmes d'exploitation généraux, les RTOS sont optimisés pour les performances en temps réel et la fiabilité, nécessaires dans des secteurs tels que l'automobile, l'aérospatial, les soins de santé, et plus.

Un RTOS est défini par sa capacité à offrir des temps de réponse prévisibles et rapides.

- **Déterminisme** : Capacité à exécuter des tâches de manière prévisible et cohérente.
- **Multitâche** : Gestion efficace de multiples tâches en parallèle.
- **Gestion des priorités** : Allocation de ressources basée sur la priorité des tâches.
- **Synchronisation des tâches** : Outils comme les mutexes, sémaphores pour coordonner les tâches.
- **Faible empreinte mémoire** : Optimisation de l'utilisation de la mémoire pour les systèmes embarqués.

Introduction aux Systèmes d'Exploitation Temps Réel (RTOS)

Différences entre RTOS et systèmes d’exploitation classiques

Les RTOS se distinguent des systèmes d'exploitation classiques (comme Windows, Linux de bureau) par leur focus sur la prévisibilité et l'efficacité dans les environnements contraints :

- **Temps de réponse** : Les RTOS répondent à des événements en temps garanti, tandis que les OS classiques visent une bonne moyenne de réponse.
- **Gestion des ressources** : Les RTOS ont des mécanismes simplifiés et plus directs pour la gestion des ressources, minimisant les délais.
- **Interface utilisateur** : Les OS classiques offrent des interfaces riches et complexes, tandis que les RTOS peuvent n'avoir aucune ou une interface très simplifiée.

Architecture et composants de FreeRTOS

1. Utilisation de FreeRTOS dans les systèmes embarqués

FreeRTOS est largement utilisé dans les systèmes embarqués pour sa simplicité et son efficacité. Il est adapté pour des applications où la fiabilité et le temps de réponse sont critiques. Sa petite taille et sa conception modulaire en font un choix idéal pour les microcontrôleurs et autres dispositifs à ressources limitées.

2. Intégration de FreeRTOS avec Yocto

Yocto est un projet qui permet de créer des systèmes Linux personnalisés pour des appareils embarqués. Bien que FreeRTOS soit un RTOS et Yocto un outil pour des systèmes basés sur Linux, il est possible d'utiliser Yocto pour gérer les dépendances et la configuration de projets qui intègrent FreeRTOS, surtout dans les systèmes complexes où cohabitent Linux et FreeRTOS sur des processeurs multicœurs.

Pour créer et configurer un projet FreeRTOS :

- **Choisissez le matériel** : Sélectionnez le processeur ou la carte sur laquelle FreeRTOS tournera.
- **Configurez FreeRTOS** : Personnalisez les options dans `FreeRTOSConfig.h` pour adapter le comportement du RTOS aux besoins spécifiques de votre application.
- **Développez vos tâches** : Programmez les tâches qui seront gérées par FreeRTOS, en définissant leurs priorités, leur pile, etc.

Intégration de FreeRTOS dans une build Yocto

Intégrer FreeRTOS dans un environnement Yocto nécessite de créer des recettes spécifiques qui définissent comment le code source de FreeRTOS est récupéré, compilé et empaqueté dans une image Yocto. Ces recettes permettent également de gérer les dépendances et les configurations spécifiques au matériel cible.

- **Créez une recette** : La recette spécifiera où télécharger FreeRTOS, comment le compiler, et où placer les fichiers résultants dans l'image finale.
- **Compilation croisée** : Configurez l'environnement pour compiler FreeRTOS pour l'architecture cible.
- **Testez sur le matériel** : Après la construction, testez le système pour s'assurer que FreeRTOS fonctionne comme prévu dans le contexte de l'image Yocto complète.