

# Sviluppo di Linux Embedded con lo Yocto Project

**Seconda Edizione**

Impara a sfruttare la potenza dello Yocto Project per creare prodotti efficienti basati su Linux



**Packt>**

[www.packt.com](http://www.packt.com)

di Otavio Salvador, Daiane Angolini

traduzione: Baldassarre Cesarano



# Sviluppo di Linux Embedded con lo Yocto Project

***Seconda Edizione***

Impara a sfruttare la potenza dello Yocto Project per creare prodotti efficienti basati su Linux

**Otavio Salvador**

**Daiane Angolini**



**BIRMINGHAM - MUMBAI**

# Sviluppo di Linux Embedded con lo Yocto Project

## *Seconda Edizione*

Copyright © 2017 Packt Publishing

Tutti i diritti riservati. Nessuna parte di questo libro può essere riprodotta, archiviata o trasmessa in qualsiasi forma o con qualsiasi mezzo, senza il previo consenso scritto dell'editore, tranne nel caso di brevi citazioni incluse in articoli critici o recensioni.

Nella preparazione di questo libro è stato compiuto ogni sforzo per garantire l'accuratezza delle informazioni presentate. Tuttavia, le informazioni contenute in questo libro sono vendute senza garanzia, esplicita o implicita. Né gli autori, né Packt Publishing, né i suoi rivenditori e distributori saranno ritenuti responsabili per eventuali danni causati o presumibilmente causati direttamente o indirettamente da questo libro.

Packt Publishing ha cercato di fornire informazioni sui marchi di tutte le società e prodotti menzionati in questo libro mediante l'uso appropriato delle maiuscole. Tuttavia, Packt Publishing non può garantire l'accuratezza di queste informazioni.

Prima pubblicazione: giugno 2014  
Seconda edizione: Novembre 2017

Riferimento di produzione: 1141117  
Pubblicato da Packt Publishing Ltd.

Livery Place  
35 Livery Street  
Birmingham  
B3 2PB, UK.

ISBN: 978-1-78847-046-9

[www.packtpub.com](http://www.packtpub.com)

# Crediti

**Autori**

Otavio Salvador

Daiane Angolini

**Revisore**

Radek Dostal

**Redazione**

Gebin George

**Editore di acquisizione**

Prateek Bharadwaj

**Editor per lo sviluppo dei contenuti**

Nikita Pawar

**Editore Tecnico**

Manish D Shanbhag

**Editore**

Safis Editing

Juliana Nair

**Coordinatore del Progetto**

Judie Jose

**Correttore di bozze**

Safis Editing

**Indice**

Rekha Nair

**Grafica**

Tanya Dutta

**Coordinamento**

Nilesh Mohite

## Gli Autori

`Otavio Salvador` ama la tecnologia e ha iniziato le sue attività col software libero nel 1999. Nel 2002, ha fondato la O.S. Systems, una società focalizzata su servizi per lo sviluppo di sistemi embedded e consulenza in tutto il mondo, creando e mantenendo BSP personalizzati e aiutando le aziende nello sviluppo dei prodotti. Questo lo ha portato a entrare a far parte della comunità OpenEmbedded nel 2008, dove è diventato un collaboratore attivo del progetto OpenEmbedded.

`Daiane Angolini` lavora con Linux embedded dal 2008. Ha lavorato come ingegnere applicativo presso NXP, occupandosi dello sviluppo interno, del porting di applicazioni custom da Android e dell'assistenza clienti in loco per le architetture i.MX in aree come kernel Linux, u-boot, Android, Yocto Project e applicazioni nello spazio utente. È, però, nello Yocto Project che ha trovato il suo posto.

## I Revisori

`Radek Dostal` è un fan di Linux e lo utilizza da 15 anni. Durante i suoi studi negli Stati Uniti, ha acquisito una passione per i sistemi embedded e da allora la combinazione di Linux con i sistemi embedded è diventata il suo pane quotidiano. Yocto ha avuto un grande impatto sul lavoro di Radek: è riuscito a convincere il suo team e i suoi manager a passare a Yocto per un progetto importante, gettando solide basi per diversi progetti successivi di successo. A Radek piace contribuire a progetti open source come parte del suo lavoro, così come durante il suo tempo libero. Se però il tempo è buono, durante il fine settimana, è molto probabile che lo troviate in montagna.

## www.PacktPub.com

Per i file allegati e i download relativi ai libri visitare [www.PacktPub.com](http://www.PacktPub.com). Sapevate che Packt offre versioni eBook di ogni libro pubblicato, con file PDF ed ePub inclusi? La versione eBook è aggiornabile su [www.PacktPub.com](http://www.PacktPub.com) e come cliente di libri cartacei, si ha diritto a uno sconto sulla copia dell'eBook. Contattateci all'indirizzo [service@packtpub.com](mailto:service@packtpub.com) per maggiori dettagli. Su [www.PacktPub.com](http://www.PacktPub.com), si possono anche leggere raccolte di articoli tecnici gratuiti, iscriversi a una serie di newsletter gratuite e ricevere sconti e offerte esclusivi su libri ed eBook Packt.



<https://www.packtpub.com/mapt>

Con Mapt si ottengono le competenze software più richieste. Mapt dà accesso completo a tutti i libri e ai videocorsi Packt, oltre agli strumenti dei leader del settore per pianificare lo sviluppo personale e avanzare in carriera.

### Perché iscriversi?

- In tutti i libri pubblicati da Packt si possono fare ricerche
- Copiare e incollare, stampare e aggiungere segnalibri al contenuto
- Su richiesta e accessibile tramite browser web

## Feedback dei lettori

Grazie per aver acquistato questo libro Packt. In Packt, la qualità è al centro del nostro processo editoriale. Per aiutarci a migliorare, lasciateci una recensione onesta sulla pagina Amazon di questo libro all'indirizzo <https://www.amazon.com/dp/178847046X>.

Per unirsi al nostro team di revisori regolari, si può inviare una e-mail a [customerreviews@packtpub.com](mailto:customerreviews@packtpub.com). Noi premiamo i nostri revisori regolari con eBook e video gratuiti in cambio del loro prezioso feedback. Aiutaci a essere inflessibili nel migliorare i nostri prodotti!

# Sommario

<b>Prefazione .....</b>	<b>11</b>
Di cosa tratta questo libro .....	11
Di cosa c'è bisogno .....	12
Per chi è questo libro .....	12
Convenzioni .....	12
Feedback dei lettori .....	13
Supporto ai clienti .....	13
Downloading delle immagini a colori di questo libro .....	13
Correzioni .....	13
Pirateria .....	13
Domande .....	13
<b>1. Introduzione allo Yocto Project .....</b>	<b>14</b>
Cos'è lo Yocto Project? .....	14
Profilo dello Yocto Project .....	14
Il ruolo di Poky .....	15
Utilizzo di BitBake .....	15
OpenEmbedded-Core .....	15
I metadati .....	15
L'alleanza tra OpenEmbedded Project e lo Yocto Project .....	16
Sommario .....	16
<b>2. Creazione del Sistema Poky-Based .....</b>	<b>17</b>
Configurazione di un sistema host .....	17
Installazione di Poky su Debian .....	17
Installazione di Poky su Fedora .....	17
Download del codice sorgente di Poky .....	18
Preparazione dell'ambiente di build .....	18
Il file local.conf .....	18
La build di un'immagine target .....	19
Esecuzione delle immagini in QEMU .....	20
Sommario .....	21
<b>3. Utilizzo di Toaster per Produrre un'immagine con BitBake .....</b>	<b>22</b>
Cos'è Toaster? .....	22
Installazione di Toaster .....	22
Avvio di Toaster .....	22
Build di un'immagine per QEMU .....	23
Sommario .....	26
<b>4. Capire il Tool BitBake .....</b>	<b>27</b>
Capire BitBake .....	27
Esplorazione dei metadati .....	27
Analisi dei metadati .....	28
Dipendenze .....	28
Input e output delle ricette .....	29
Prelievo del codice sorgente .....	30
Download di file remoti .....	30
I repository Git .....	31
Ottimizzazione del download dei sorgenti .....	31
Disabilitare l'accesso alla rete .....	32
Il ruolo di BitBake .....	32
Estensione dei task .....	33
Generazione di un'immagine del filesystem di root .....	33
Sommario .....	35
<b>5. Dettagli sulla Directory Temporanea di Build .....</b>	<b>36</b>
Dettagliare la build directory .....	36
Costruzione della build directory .....	36
Esplorazione della directory di build temporanea .....	37
Capire la directory di lavoro .....	37
Capire le directory sysroot .....	39
Sommario .....	40



<b>6. Il supporto al Packaging</b>	<b>41</b>
Utilizzo dei formati di package supportati	41
Elenco dei formati di package supportati	41
Scelta del formato del pacchetto	41
Codice in esecuzione durante l'installazione del pacchetto	42
La cache di stato condivisa	43
Il versioning del package	44
La specifica delle dipendenze del pacchetto di runtime	44
Feed dei pacchetti	45
Utilizzo dei feed dei pacchetti	46
Sommario	47
<b>7. Approfondimenti sui Metadati di BitBake</b>	<b>48</b>
Utilizzo dei metadati	48
Lavorare con i metadati	48
L'impostazione della variabile di base	48
Espansione della variabile	49
Impostazione di un valore di default utilizzando ?=	49
Impostazione di un valore di default utilizzando ??=	49
Espansione immediata della variabile	49
Accodare e anteporre	50
Sintassi degli operatori di override	50
Set di metadati condizionali	50
Accodamento condizionale	51
Inclusione di file	51
Espansione delle variabili Python	51
Definizione dei metadati eseguibili	51
Definizione delle funzioni Python nello spazio dei nomi globale	52
Il sistema dell'ereditarietà	52
Sommario	52
<b>8. Sviluppo con lo Yocto Project</b>	<b>53</b>
Decifrare il "software development kit"	53
Lavorare con l'SDK di Poky	53
Utilizzo di un SDK basato su immagini	54
SDK generico – meta-toolchain	54
Utilizzo di un SDK	55
Sviluppo di applicazioni sul target	55
Integrazione con Eclipse	56
Sommario	57
<b>9. Debugging con lo Yocto Project</b>	<b>58</b>
Differenziazione dei metadati e del debug delle applicazioni	58
Tracking dell'immagine, pacchetti e contenuto dell'SDK	58
Debugging dei pacchetti	59
Le informazioni dei log durante l'esecuzione dei task	60
Utilizzo di una shell di sviluppo	60
Utilizzo di GNU Project Debugger per il debug	61
Sommario	62
<b>10. Esplorazione dei Layer Esterni</b>	<b>63</b>
Potenziare la flessibilità con i layer	63
Dettagliare il codice sorgente del layer	64
Aggiunta di meta layer	65
L'ecosistema di layer dello Yocto Project	66
Sommario	67
<b>11. Creazione di Layer Personalizzati</b>	<b>68</b>
Creare un nuovo layer	68
Aggiunta di metadati al layer	69
Creazione di un'immagine	69
Aggiunta di una ricetta del pacchetto	70
Creazione automatica di una ricetta del pacchetto base utilizzando "recipetool"	71
Aggiunta del supporto a una nuova definizione di macchina	73
Confezionare un'immagine per la macchina	74
Utilizzo di una distribuzione personalizzata	74

MACHINE_FEATURES e DISTRO_FEATURES.....	76
Lo 'scope' delle variabili.....	76
Sommario.....	76
<b>12. Personalizzazione di ricette esistenti .....</b>	<b>77</b>
Casi d'uso comuni .....	77
Aggiunta di opzioni extra alle ricette basate su Autoconf .....	77
Applicazione di una patch .....	78
Aggiunta di file extra ai pacchetti esistenti .....	78
Il path di ricerca dei file .....	78
Modifica della configurazione della funzione della ricetta .....	79
Personalizzazione di BusyBox .....	79
Personalizzazione del framework linux-yocto .....	80
Sommario.....	80
<b>13. La conformità GPL .....</b>	<b>81</b>
Il copyleft.....	81
Conformità al copyleft rispetto al codice proprietario .....	81
Alcune linee guida per la conformità delle licenze .....	81
Gestione delle licenze software con Poky .....	82
Licenze commerciali.....	82
Utilizzo di Poky per ottenere la conformità con il copyleft .....	83
Controllo delle licenze .....	83
Fornire il codice sorgente .....	83
Fornitura di script di compilazione e modifiche al codice sorgente.....	84
Fornire il testo della licenza.....	85
Sommario.....	85
<b>14. Avvio di Linux Embedded Custom .....</b>	<b>86</b>
Esplorazione delle schede .....	86
Alla scoperta del livello BSP giusto .....	87
Il Baking per l'hardware.....	87
Baking per BeagleBone Black.....	88
Baking per Raspberry Pi 3 .....	88
Baking per la Wandboard.....	89
Boot dell'immagine .....	90
Boot di BeagleBone Black dalla scheda SD .....	90
Boot di Raspberry Pi 3 dalla scheda SD .....	90
Boot di Wandboard dalla scheda SD .....	91
Passi successivi .....	91
Sommario.....	91

# Prefazione

Col trend attuale, Linux è la grande novità. Linux ha costantemente rilasciato prodotti open source all'avanguardia e i Sistemi Embedded sono stati inseriti nel bagaglio tecnologico dell'umanità.

Lo Yocto Project è in una ottima posizione come scelta per i progetti e fornisce un ricco set di strumenti per utilizzare la maggior parte delle energie e risorse nello sviluppo del prodotto, invece di reinventare la ruota.

I soliti compiti e requisiti dei prodotti e dei team di sviluppo basati su Linux Embedded sono stati la linea guida per il concepimento di questo libro. Essendo scritto da membri attivi della comunità, con un approccio pratico e diretto, è un trampolino di lancio sia per la curva di apprendimento che per il progetto del prodotto.

## ***Di cosa tratta questo libro***

Capitolo 1, *Primo incontro con lo Yocto Project*, presenta i primi concetti e le premesse per introdurre le parti dello Yocto Project e gli strumenti principali.

Capitolo 2, *Creazione del Sistema Poky-Based*, presenta l'ambiente necessario per la prima build.

Capitolo 3, *Uso di Toaster per Creare un'Immagine*, mostra la comoda interfaccia web utilizzabile come wrapper per la configurazione e come strumento per il build.

Capitolo 4, *Capire il Tool BitBake*, presenta il tool BitBake e come gestisce i task e le dipendenze.

Capitolo 5, *Dettagli sulla Directory Temporanea di Build*, descrive in dettaglio la cartella temporanea di output di una build.

Capitolo 6, *Il Supporto al Packaging*, spiega il meccanismo dei pacchetti utilizzato come base per creare e gestire tutti i pacchetti dei binari.

Capitolo 7, *Approfondimenti sui Metadati di BitBake*, descrive in dettaglio il linguaggio dei metadati di BitBake utilizzato in tutti gli altri capitoli.

Capitolo 8, *Lo Sviluppo con lo Yocto Project*, mostra il flusso di lavoro per ottenere un ambiente di sviluppo.

Capitolo 9, *Il Debugging con lo Yocto Project*, mostra come usare Poky per generare ed usare un ambiente di debug.

Capitolo 10, *Esplorazione dei Layer Esterni*, esplora uno dei concetti più importanti dello Yocto Project—la flessibilità dell'uso dei layer esterni.

Capitolo 11, *Creazione di un Layer Custom*, esercizi pratici sulla creazione di layer.

Capitolo 12, *Personalizzazione delle Ricette Esistenti*, esempi su come personalizzare le ricette esistenti.

Capitolo 13, *source poky/oe-init-build-env*, riassume le attività e i concetti per un prodotto conforme al copyleft.

Capitolo 14, *Avvio di Linux Embedded Custom*, uso di macchine hardware reali con i tool dello Yocto Project.

## Di cosa c'è bisogno

Per seguire meglio questo libro, è importante che si abbia un background pregresso su alcuni argomenti che non sono trattati o sono solo brevemente menzionati nel testo, come la conoscenza generale di Git, e del kernel Linux nonché le basi del processo di compilazione.

Per capire il quadro generale dello Yocto Project prima di passare ai dettagli dei concetti tecnici, consigliamo l'opuscolo open source, *Heading for the Yocto Project*, che si trova sotto il repository Git <https://git.io/vFUiI>, il contenuto di questo opuscolo ha lo scopo di aiutare i nuovi arrivati ad acquisire una migliore comprensione degli obiettivi dello Yocto Project e dei suoi potenziali usi e intende fornire una panoramica del progetto, prima di immergersi nei dettagli tecnici su come le cose possono essere fatte.

È richiesta una conoscenza minima sull'uso dell'ambiente GNU/Linux e di Linux Embedded, nonché quella dei concetti generali utilizzati nello sviluppo come la compilazione, il debugging, la distribuzione e l'installazione. Una certa esperienza con Shell Script e Python è un vantaggio, perché tali linguaggi di programmazione sono tecnologie di base ampiamente utilizzate dagli strumenti dello Yocto Project.

La mancanza dei concetti elencati sopra, non dev'essere un deterrente, ma come qualcosa che si può imparare e, allo stesso tempo, utilizzare, con questo libro. Tuttavia, se si vuol saperne di più su questi argomenti, consigliamo il libro *Mastering Embedded Linux Programming*, ISBN: 9781787283282, di *Chris Simmonds*.

Qualsiasi concetto citato sopra non dovrebbe scoraggiare la lettura di questo libro perché sono conoscenze che possono essere apprese contemporaneamente.

## Per chi è questo libro

Questo libro è destinato a ingegneri e appassionati con un'esperienza di Linux Embedded, desiderosi di apprendere gli strumenti dello Yocto Project per la valutazione, il confronto o l'uso in un progetto. Questo libro ha lo scopo di preparare rapidamente impedendo di rimanere intrappolati nelle solite insidie della curva di apprendimento.

## Convenzioni

In questo libro si troveranno molti stili di testo che separano i diversi tipi di informazioni. Ecco alcuni esempi di questi stili e una spiegazione del loro significato.

Un blocco di codice è impostato come segue:

```
[default]
T = "123"
A := "${B} ${A} test ${T}"
T = "456"
B = "${T} bval"
C = "cval"
C := "${C}append"
```

Quando si vuole attirare l'attenzione del lettore su qualche particolare parte di un blocco di codice, le righe o gli elementi rilevanti vengono riportati in grassetto:

```
[default]
T = "123"
A := "${B} ${A} test ${T}"
T = "456"
B = "${T} bval"
C = "cval"
C := "${C}append"
```

Qualsiasi input o output sulla riga di comando viene scritto come segue:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo build-essential chrpath
```

I nuovi termini e le parole importanti sono in grassetto. Le parole che si vedono sullo schermo, ad esempio, nei menù o nelle finestre di dialogo, vengono visualizzate nel testo in questo modo: "Dopodiché, cliccare sulla scheda 'Image Recipes' per scegliere l'immagine da creare."

Avvisi o note importanti vengono visualizzati in una casella come questa.

*Suggerimenti e trucchi appaiono in questo modo.*

## Feedback dei lettori

Il feedback dei nostri lettori è sempre il benvenuto. Fateci sapere cosa ne pensate di questo libro, cosa è piaciuto o cosa no. Il feedback dei lettori è importante per noi per sviluppare titoli dai quali si otterrà il massimo.

Per inviarci un feedback generale, è sufficiente inviare un'e-mail a [feedback@packtpub.com](mailto:feedback@packtpub.com) e menzionare il titolo del libro nell'oggetto del messaggio.

Se c'è un argomento in cui si è esperti e si è interessati a scrivere o contribuire a un libro, consultare la nostra guida all'autore su [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Supporto ai clienti

Ora che si è l'orgoglioso proprietario di un libro Packt, abbiamo una serie di cose per avere il massimo dall'acquisto.

## Downloading delle immagini a colori di questo libro

Forniamo anche un file PDF con immagini a colori degli screenshot/diagrammi utilizzati in questo libro. Le immagini a colori aiuteranno a capire meglio le modifiche nell'output. Si può scaricare da [https://www.packtpub.com/sites/default/files/downloads/EmbeddedLinuxDevelopmentusingYoctoProjectsSecondEdition\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/EmbeddedLinuxDevelopmentusingYoctoProjectsSecondEdition_ColorImages.pdf).

## Correzioni

Anche se abbiamo posto ogni cura per garantire l'accuratezza dei contenuti, gli errori capitano. Se si trova un errore in uno dei libri, forse un errore nel testo o nel codice, saremmo grati se lo segnalaste. In questo modo, si può evitare la frustrazione ad altri lettori e ci aiuta a migliorare le versioni successive di questo libro. Se si trovano errori, segnalarli visitando <http://www.packtpub.com/submit-errata>, selezionando il libro, cliccando sul link "Errata Submission Form" e inserendo i dettagli dell'errore. Dopo le verifiche, la richiesta sarà accettata e l'errore verrà caricato sul nostro sito Web o aggiunto a qualsiasi elenco di errata esistenti nella sezione Errata di quel titolo. Tutte le correzioni esistenti sono visualizzabili selezionando il proprio titolo da <http://www.packtpub.com/support>.

## Pirateria

La pirateria del materiale protetto da copyright su Internet è un problema continuo su tutti i media. In Packt, prendiamo molto sul serio la protezione dei nostri diritti d'autore e delle nostre licenze. Se ci si imbatte in copie illegali delle nostre opere in qualsiasi forma su Internet, vi preghiamo di fornirci immediatamente l'indirizzo della posizione o il nome del sito Web in modo che possiamo perseguire per vie legali.

Si prega di contattare l'indirizzo [copyright@packtpub.com](mailto:copyright@packtpub.com) con un link al materiale sospetto illegale.

Apprezziamo l'aiuto nel proteggere i nostri autori e la nostra capacità di offrire contenuti di valore.

## Domande

Ci potete contattare all'indirizzo [questions@packtpub.com](mailto:questions@packtpub.com) per qualsiasi problema riguardo al libro, e faremo del nostro meglio per risolverlo.

# 1. Introduzione allo Yocto Project

In questo capitolo verrà presentato lo `Yocto Project`. Qui, vengono discussi i concetti principali del progetto, costantemente utilizzati in tutto il libro. Vedremo in breve la storia dello Yocto Project, di OpenEmbedded, di Poky, di BitBake e dei metadati, da qui, ci si allaccia la cintura di sicurezza e si parte!

## **Cos'è lo Yocto Project?**

Lo Yocto Project è un gruppo di lavoro della Linux Foundation definito come:

*"Lo Yocto Project fornisce infrastruttura e strumenti open source di alta qualità per aiutare gli sviluppatori a creare le proprie distribuzioni Linux personalizzate per qualsiasi architettura hardware, in più segmenti di mercato. Lo Yocto Project ha lo scopo di fornire un utile punto di partenza per gli sviluppatori".*

Lo Yocto Project è un progetto di collaborazione open source che fornisce template, strumenti e metodi per creare sistemi personalizzati basati su Linux per prodotti embedded indipendentemente dall'architettura hardware. Essendo gestito da un collega della Linux Foundation, il progetto rimane indipendente dalle organizzazioni membri che partecipano in vari modi fornendo risorse al progetto.

È stato fondato nel 2010 dalla collaborazione di molti produttori di hardware, sistemi operativi open source, fornitori e società di elettronica nel tentativo di ridurre la duplicazione del lavoro, fornire risorse e informazioni per utenti nuovi ed esperti.

Tra queste risorse c'è OpenEmbedded-Core, il componente principale del sistema, fornito dal progetto OpenEmbedded.

Lo Yocto Project è, quindi, un progetto di comunità open source che aggrega diverse aziende, comunità, progetti e strumenti, riunendo persone con lo stesso scopo per costruire prodotti embedded basati su Linux; tutti questi componenti sono raggruppati assieme, essendo guidati dalla comunità che ne ha bisogno per lavorare.

## **Profilo dello Yocto Project**

Per comprendere i compiti e i risultati dallo Yocto Project, possiamo usare l'analogia di una macchina informatica. L'input è un insieme di dati che descrive ciò che vogliamo, ovvero la nostra specifica. Come output, abbiamo il prodotto basato su Linux embedded.

Se l'output è un prodotto che esegue un sistema operativo basato su Linux, il risultato generato sono i pezzi che compongono il sistema operativo, come il kernel Linux, il bootloader e il bundle del filesystem di root (`rootfs`), correttamente organizzati.

Per produrre il bundle di `rootfs` risultante e altri prodotti finali, i tool dello Yocto Project intervengono in tutti i passaggi intermedi. Il riutilizzo di tool e altri componenti software creati in precedenza viene massimizzato durante la creazione di altre applicazioni, librerie e qualsiasi altro componente software nell'ordine corretto e con la configurazione desiderata, incluso il prelievo del codice sorgente richiesto dai rispettivi repository, come The Linux Kernel Archives ([www.kernel.org](http://www.kernel.org)), GitHub, e [www.SourceForge.net](http://www.SourceForge.net).

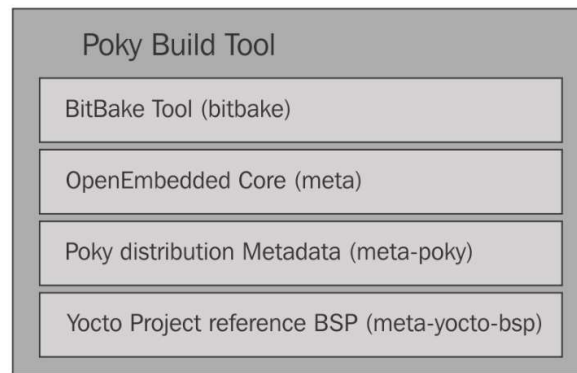
I tool dello Yocto Project preparano il proprio ambiente di build, le utilità e la toolchain, riducendo la quantità di dipendenza dal software dell'host. Un'implicazione subdola ma importante è che il determinismo viene considerevolmente aumentato poiché i tool e le loro versioni così come le loro opzioni di configurazione sono le stesse, riducendo così al minimo il numero di utilità host su cui fare affidamento e producendo lo stesso risultato indipendentemente dalla distribuzione Linux usata sull'host.

Possiamo elencare alcuni progetti, come Poky, BitBake e OpenEmbedded-Core, sotto l'ombrello dello Yocto Project, tutti complementari e che svolgono ruoli specifici nel sistema. Capiremo esattamente come lavorano insieme in questo capitolo e in tutto il libro.

## ***Il ruolo di Poky***

Poky è il sistema di riferimento dello Yocto Project ed è composto da una raccolta di tool e metadati. È indipendente dalla piattaforma ed esegue la cross-compilazione, utilizzando BitBake, OpenEmbedded Core e un set predefinito di metadati, come mostrato nella figura seguente. Fornisce il meccanismo per creare e combinare migliaia di progetti open source distribuiti per formare uno stack software Linux completamente personalizzabile, completo e coerente.

L'obiettivo principale di Poky è quello di fornire tutte le funzionalità di cui uno sviluppatore embedded ha bisogno.



## ***Utilizzo di BitBake***

BitBake è un'utilità di pianificazione che analizza sia il codice di Python che gli script shell. Il codice analizzato genera ed esegue task, che sono fondamentalmente un insieme di passaggi ordinati in base alle dipendenze del codice.

Valuta tutti i file di configurazione e i dati delle ricette disponibili (noti come metadati), gestendo l'espansione delle variabili dinamiche, le dipendenze e la generazione del codice. Tiene traccia di tutte le attività in elaborazione al fine di garantirne il completamento, massimizzando l'uso delle risorse di elaborazione per ridurre i tempi di build ed essere predicibile. Lo sviluppo di BitBake è centralizzato nella mailing list `bitbake-devel@lists.openembedded.org` e il suo codice si trova nella sottodirectory `bitbake` di Poky.

## ***OpenEmbedded-Core***

La collezione di metadati OpenEmbedded-Core fornisce il motore del tool di compilazione Poky. È progettata per fornire le funzionalità principali e per essere la più snella possibile. Fornisce il supporto per sette diverse architetture di processori (ARM, ARM64, x86, x86-64, PowerPC, MIPS e MIPS64), supportando solo le schede che devono essere emulate da QEMU.

Lo sviluppo è centralizzato nella mailing `openembedded-core@lists.openembedded.org` e ospita i suoi metadati all'interno della `meta` sottodirectory di Poky.

## ***I metadati***

I metadati, composti da un insieme di file Python e Shell Script, compongono un sistema estremamente flessibile. Poky lo usa per estendere OpenEmbedded-Core e comprende due diversi livelli, che sono altri sottoinsiemi di metadati mostrati in questo modo:

- **meta-poky**: Questo layer fornisce i default e le distribuzioni supportate, il logo e le informazioni sul tracciamento dei metadati (manutentori, stato dell'upstream, ecc.)
- **meta-yocto-bsp**: Fornisce il Board Support Package (BSP) utilizzato come riferimento per lo sviluppo dello Yocto Project e per il processo di Quality Assurance (QA)

Capitolo 8, *Lo Sviluppo con lo Yocto Project*, esplora più dettagliatamente i metadati e funge da riferimento per la scrittura delle ricette.

## ***L'alleanza tra OpenEmbedded Project e lo Yocto Project***

Il progetto OpenEmbedded è stato creato intorno al gennaio del 2003 quando alcuni sviluppatori del progetto OpenZaurus hanno iniziato a lavorare con il nuovo sistema di build. Il sistema di build OpenEmbedded è stato, sin dal suo inizio, un "task scheduler" ispirato e basato sul sistema di pacchetti Gentoo Portage chiamato BitBake. Il progetto ha ampliato la sua collezione di software e supportando velocemente insieme di macchine.

Come conseguenza dello sviluppo non coordinato, era difficile utilizzare OpenEmbedded in prodotti che richiedevano una base di codice più stabile e pulita, motivo per cui è nato Poky. Poky è partito come un sottoinsieme di OpenEmbedded e aveva una base di codice più pulita e stabile su un insieme limitato di architetture. Le sue dimensioni ridotte gli hanno permesso di iniziare a sviluppare tecnologie emergenti, come i plug-in IDE e l'integrazione in QEMU, che sono ancora utilizzate oggi.

Intorno a novembre del 2010, è stato annunciato dalla Linux Foundation lo Yocto Project per continuare questo lavoro nell'ambito di un progetto sponsorizzato dalla Linux Foundation. Lo Yocto Project e OpenEmbedded Project hanno consolidato i loro sforzi su un sistema di build di base chiamato OpenEmbedded-Core, utilizzando il meglio di Poky e di OpenEmbedded, enfatizzando così un maggiore utilizzo di altri componenti, metadati e sottoinsiemi.

## ***Sommario***

Questo primo capitolo ha fornito una panoramica di come il OpenEmbedded Project è correlato allo Yocto Project, i componenti che formano Poky e come è stato creato. Nel prossimo capitolo, verrà presentato il flusso di lavoro di Poky con i passaggi per scaricare, configurare e preparare l'ambiente di build Poky e come creare e far funzionare la prima immagine utilizzando QEMU.



## 2. Creazione del Sistema Poky-Based

In questo capitolo capiremo i concetti di base coinvolti nel flusso di lavoro di Poky. Sporchiamoci le mani con i passaggi per scaricare e configurare, preparare l'ambiente di build Poky e creare qualcosa di utilizzabile. I passaggi qui trattati sono normalmente usati per il test e lo sviluppo. Forniscono l'intera esperienza di utilizzo di Poky e un assaggio delle sue capacità.

### **Configurazione di un sistema host**

Il processo da impostare sul sistema host dipende dalla distribuzione che eseguiamo su di esso. Poky supporta una serie di distribuzioni Linux e, se si è all'inizio nello sviluppo di Linux embedded, è consigliabile utilizzare una delle distribuzioni Linux supportate per evitare di perdere tempo a eseguire il debug dei problemi di build relativi al supporto del sistema host.

La versione corrente di una delle seguenti distribuzioni, dovrebbe andare bene:

- Ubuntu
- Fedora
- CentOS
- Debian
- openSUSE

Per avere conferma che la propria versione sia supportata, si consiglia di controllare sulla documentazione ufficiale online <http://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html#detailed-supported-distros>.

Se la distribuzione non è nell'elenco precedente, non significa che non sia possibile utilizzarci Poky. Tuttavia, non si sa se funzionerà e si potrebbero avere problemi.

I pacchetti da installare nel sistema host variano da una distribuzione all'altra. In questo libro si troveranno le istruzioni per *Debian* e *Fedora*, le nostre distribuzioni preferite. Si possono trovare le istruzioni per tutte le distribuzioni supportate nel *Manuale di riferimento dello Yocto Project*.

### **Installazione di Poky su Debian**

Per installare i pacchetti necessari per un sistema host 'headless' [senza grafica], eseguire il comando seguente:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential  
chrpath socat cpio python python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping
```

Se il sistema host dispone del supporto per la grafica, eseguire il comando seguente:

```
$ sudo apt-get install libsdl1.2-dev xterm
```

I comandi precedenti sono anche compatibili con le distribuzioni Ubuntu.

### **Installazione di Poky su Fedora**

Per installare i pacchetti necessari per un sistema host 'headless' [senza grafica], eseguire il comando seguente:

```
$ sudo dnf install gawk make wget tar bzip2 gzip python3 unzip perl patch diffutils diffstat git cpp  
gcc gcc-c++ glibc-devel texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-  
Queue perl-bignum socat python3-pexpect findutils which file cpio python python3-pip xz which
```

Se il sistema host dispone del supporto per la grafica, eseguire il comando seguente:

```
$ sudo yum install SDL-devel xterm
```

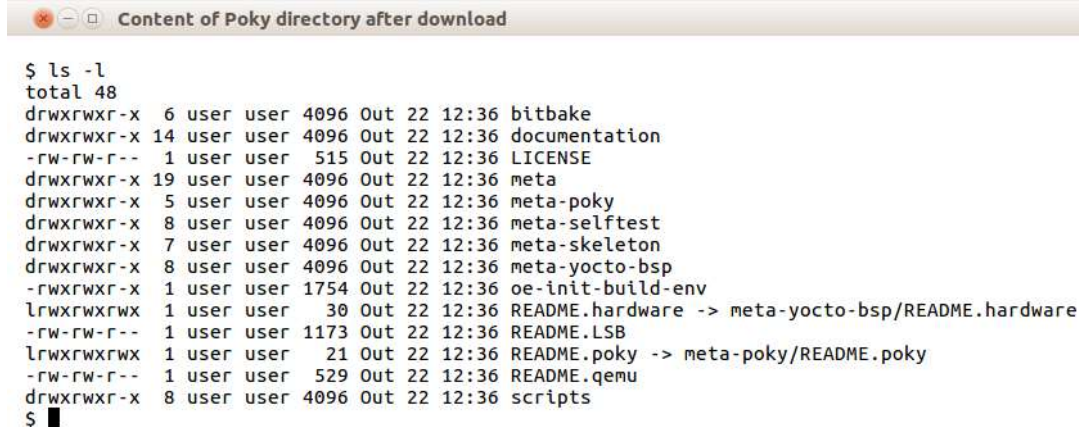
## Download del codice sorgente di Poky

Dopo aver installato i pacchetti necessari nel sistema host di sviluppo, si deve ottenere il codice sorgente di Poky scaricabile con Git col seguente comando:

```
$ git clone git://git.yoctoproject.org/poky -b rocko
```

Maggiori informazioni su Git: <http://git-scm.com>.

Al termine del download, la directory poky dovrebbe avere i seguenti contenuti:



```
$ ls -l
total 48
drwxrwxr-x 6 user user 4096 Out 22 12:36 bitbake
drwxrwxr-x 14 user user 4096 Out 22 12:36 documentation
-rw-rw-r-- 1 user user 515 Out 22 12:36 LICENSE
drwxrwxr-x 19 user user 4096 Out 22 12:36 meta
drwxrwxr-x 5 user user 4096 Out 22 12:36 meta-poky
drwxrwxr-x 8 user user 4096 Out 22 12:36 meta-selftest
drwxrwxr-x 7 user user 4096 Out 22 12:36 meta-skeleton
drwxrwxr-x 8 user user 4096 Out 22 12:36 meta-yocto-bsp
-rwxrwxr-x 1 user user 1754 Out 22 12:36 oe-init-build-env
lrwxrwxrwx 1 user user 30 Out 22 12:36 README.hardware -> meta-yocto-bsp/README.hardware
-rw-rw-r-- 1 user user 1173 Out 22 12:36 README.LSB
lrwxrwxrwx 1 user user 21 Out 22 12:36 README.poky -> meta-poky/README.poky
-rw-rw-r-- 1 user user 529 Out 22 12:36 README.qemu
drwxrwxr-x 8 user user 4096 Out 22 12:36 scripts
$
```

Gli esempi e il codice presentati in questo e nei capitoli seguenti utilizzano lo Yocto Project Version 2.4. Il nome in codice è Rocko, come riferimento.

## Preparazione dell'ambiente di build

Nella directory poky, c'è lo script `oe-init-build-env`, utilizzabile per configurare l'ambiente di build. Lo script va eseguito in questo modo:

```
$ source oe-init-build-env [build-directory]
```

Qui, `build-directory` è un parametro opzionale per il nome della directory in cui è impostato l'ambiente; nel caso in cui non venga fornito, il default è `build`. La `build-directory` è dove si eseguono le build.

È molto comodo utilizzare diverse directory di build. Si può lavorare su progetti distinti in parallelo o con diverse configurazioni sperimentali senza influenzare le altre build.

In tutto il libro, si userà `build` come directory di build. Quando si deve puntare a un file all'interno della directory `build`, adotteremo la stessa convenzione, ad esempio, `build/conf/local.conf`.

## Il file local.conf

Quando si inizializza un ambiente di build, viene creato un file chiamato `build/conf/local.conf`, che è un potente strumento in grado di configurare quasi ogni aspetto del processo di build. Si può impostare la macchina per la quale si esegue la build, l'architettura host della toolchain da utilizzare per una cross-toolchain personalizzata, ottimizzare le opzioni per la riduzione massima dei tempi di build e così via. I commenti all'interno del file `build/conf/local.conf` costituiscono un'ottima documentazione e un riferimento di possibili variabili e dei rispettivi default. L'insieme minimo di variabili che probabilmente si vorrà modificare rispetto al default è il seguente:

```
MACHINE ??= "qemux86"
```

La variabile **MACHINE** è quella che determina la macchina target per la quale si esegue la build. Al momento della stesura di questo libro, Poky supporta le seguenti macchine nel suo BSP di riferimento:

- **beaglebone**: Questa è BeagleBone, la piattaforma di riferimento per ARM a 32 bit

- `genericx86`: Questo è un supporto generico per macchine basate su x86 a 32 bit
- `genericx86-64`: Questo è un supporto generico per macchine basate su x86 a 64 bit
- `mpc8315e-rdb`: Questa è una piattaforma di riferimento NXP MPC8315 PowerPC
- `edgerouter`: Questa è EdgeRouter Lite, ovvero la piattaforma di riferimento 64-bit MIPS

Le macchine sono rese disponibili da un layer chiamato `meta-yocto-bsp`. Oltre a queste macchine, OpenEmbedded-Core fornisce anche il supporto per:

- `qemuarm`: Questa è l'emulazione QEMU di ARM
- `qemuarm64`: Questa è l'emulazione QEMU di ARM64
- `qemumips`: Questa è l'emulazione QEMU di MIPS
- `qemumips64`: Questa è l'emulazione QEMU di MIPS64
- `qemuppc`: Questa è l'emulazione QEMU di PowerPC
- `qemux86-64`: Questa è l'emulazione QEMU di x86-64
- `qemux86`: Questa è l'emulazione QEMU di x86

Altre macchine sono supportate tramite layer BSP aggiuntivi, provenienti da diversi fornitori. Il procedimento per utilizzare un layer BSP aggiuntivo è mostrato nel [Capitolo 10, Esplorazione di Layer Esterni \[da altre fonti\]](#).

Il file `local.conf` è un modo molto conveniente per sovrascrivere diverse configurazioni di default per tutti i tool dello Yocto Project. In sostanza, si può modificare o impostare qualsiasi variabile, ad esempio aggiungere ulteriori pacchetti a un file immagine. Sebbene sia conveniente, dovrebbe essere considerato come una modifica temporanea poiché il file `build/conf/local.conf` di solito non viene tracciato da alcun sistema di gestione del codice sorgente.

Ci sono diverse variabili impostabili nel file `build/conf/local.conf`. Vale la pena dedicare del tempo e leggere i commenti nel file generato per avere un'idea generale.

## La build di un'immagine target

Poky fornisce diverse ricette predefinite di immagini utilizzabili per creare un'immagine binaria. Possiamo controllare l'elenco delle immagini disponibili eseguendo il seguente comando dalla directory `poky`:

```
$ ls meta*/recipes*/images/*.bb
```

Tutte le ricette producono immagini che sono un insieme di pacchetti decompressi e configurati, generando un filesystem utilizzabile su un hardware o su una delle macchine QEMU supportate.

Di seguito, c'è l'elenco delle immagini più comunemente utilizzate:

- `core-image-minimal`: Questa è una piccola immagine che consente il boot [avvio] di un dispositivo ed è molto utile per i test e lo sviluppo del kernel e del boot loader.
- `core-image-base`: Questa è un'immagine "console-only" che supporta completamente l'hardware del dispositivo di target.
- `core-image-weston`: Questa è un'immagine che fornisce le librerie del protocollo Wayland e il compositore Weston di riferimento.
- `core-image-x11`: Questa è un'immagine X11 molto semplice con un terminale.
- `core-image-sato`: Questa è un'immagine con supporto Sato e un ambiente mobile per dispositivi mobili che utilizzano X11; fornisce applicazioni come un terminale, un editor, un file manager, un lettore multimediale e così via.

L'elenco completo sarebbe probabilmente obsoleto, quindi non è incluso. Esistono diverse immagini che supportano diverse funzionalità come Real Time, InitRAMFS, MTD (strumenti flash) e altre. Si consiglia di controllare il codice sorgente o il *Manuale di riferimento dello Yocto Project* per l'elenco completo e aggiornato.

Il processo di build di un'immagine per un target è molto semplice. Si deve lanciare il seguente comando:

```
$ bitbake <recipe name>
```

Nel seguente esempio si userà MACHINE = "qemuarm". Dovrebbe essere impostato in build/conf/local.conf di conseguenza.

Ad esempio, per il build di `core-image-full-cmdline`, si esegue il comando seguente:

```
$ bitbake core-image-full-cmdline
```

## ***Esecuzione delle immagini in QEMU***

Dato che molti progetti dipendono dall'hardware, l'emulazione hardware accelera il processo di sviluppo, consentendo un'esecuzione di test senza coinvolgere alcun hardware reale.

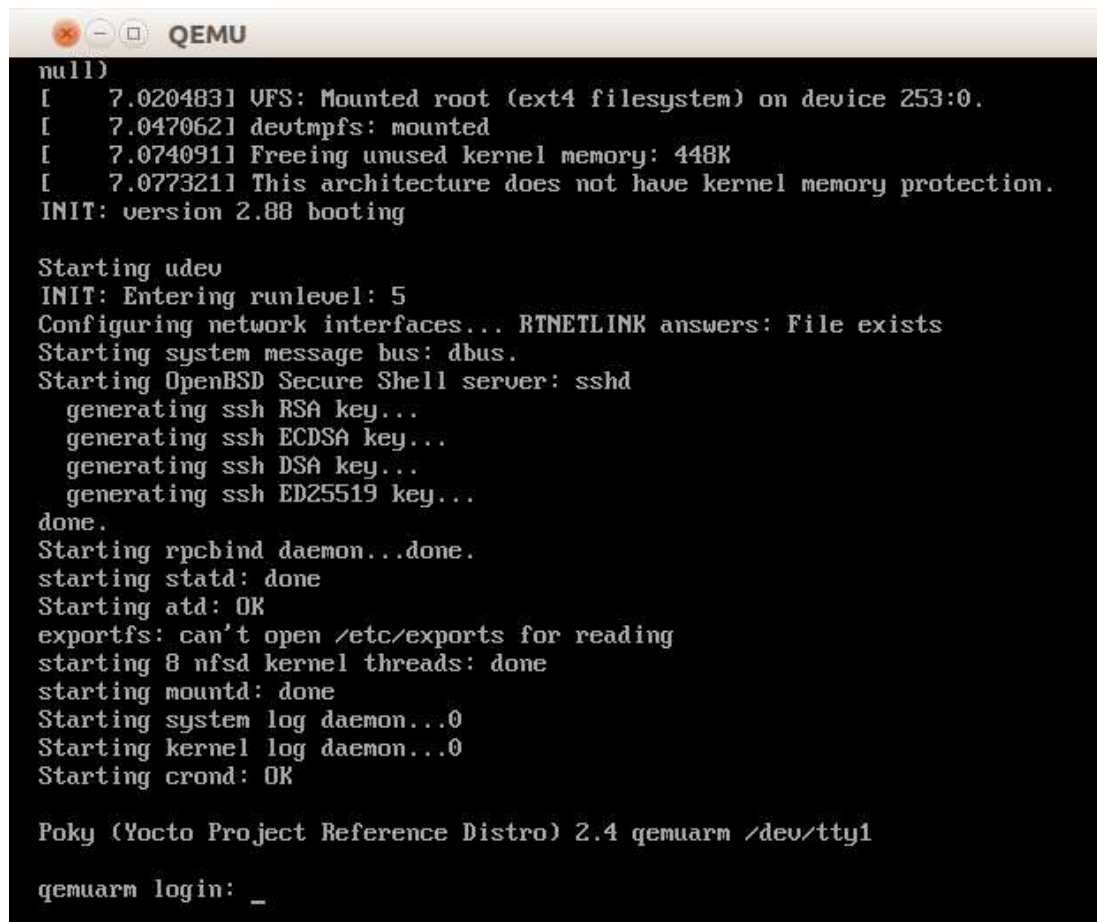
Quick EMULATOR (QEMU) è un pacchetto software gratuito e open source che esegue la virtualizzazione dell'hardware. Le macchine basate su QEMU consentono i test e lo sviluppo senza un hardware reale. Attualmente sono supportate le emulazioni ARM, ARM64, MIPS, MIPS64, PowerPC, x86 e x86-64.

Lo script `runqemu` abilita e semplifica l'uso di QEMU con le macchine supportate da OpenEmbedded-Core. Il modo per eseguire lo script è il seguente:

```
$ runqemu <machine> <zimage> <filesystem>
```

Qui, `<machine>` è la macchina/architettura da utilizzare, come `qemuarm`, `qemumips`, `qemuppc`, `qemux86` o `qemux86-64`. Inoltre, `<zimage>` è il percorso di un kernel (ad esempio, `zimage-qemuarm.bin`). Infine, `<filesystem>` è il path di un'immagine `ext3` (ad esempio, `filesystem-qemuarm.ext3`) o di una directory NFS. I parametri `<zimage>` e `<filesystem>` sono opzionali.

Quindi, ad esempio, nel caso in cui si esegua `runqemu qemuarm core-image-full-cmdline`, si può vedere qualcosa di simile a quanto mostrato nella seguente schermata:



```
QEMU
null)
[ 7.020483] UFS: Mounted root (ext4 filesystem) on device 253:0.
[ 7.047062] devtmpfs: mounted
[ 7.074091] Freeing unused kernel memory: 448K
[ 7.077321] This architecture does not have kernel memory protection.
INIT: version 2.88 booting

Starting udev
INIT: Entering runlevel: 5
Configuring network interfaces... RTNETLINK answers: File exists
Starting system message bus: dbus.
Starting OpenBSD Secure Shell server: sshd
    generating ssh RSA key...
    generating ssh ECDSA key...
    generating ssh DSA key...
    generating ssh ED25519 key...
done.
Starting rpcbind daemon...done.
starting statd: done
Starting atd: OK
exportfs: can't open /etc/exports for reading
starting 8 nfsd kernel threads: done
starting mountd: done
Starting system log daemon...0
Starting kernel log daemon...0
Starting crond: OK

Poky (Yocto Project Reference Distro) 2.4 qemuarm /dev/tty1

qemuarm login: _
```

Si può effettuare il login come root con la password vuota. Il sistema si comporta come un normale sistema anche se utilizzato all'interno di QEMU. Il processo per distribuire un'immagine in un hardware reale varia a seconda del tipo di storage utilizzato, del bootloader e così via. Il processo per generare l'immagine, però, è lo stesso. Nel [Capitolo 14, Avvio del Linux Embedded Custom](#), viene discusso come creare ed eseguire un'immagine nelle macchine BeagleBone Black, Raspberry Pi 3 e Wandboard.

## Sommario

In questo capitolo abbiamo appreso i passaggi necessari per configurare Poky e creare la prima immagine. È stata eseguita quell'immagine utilizzando `runqemu`, fornendo una buona panoramica delle capacità disponibili. Nel prossimo capitolo viene presentato Toaster, che fornisce un'interfaccia "user friendly" per BitBake, e verrà usata per la build di un'immagine e per personalizzarla ulteriormente.

## 3. Utilizzo di Toaster per Produrre un'immagine con BitBake

Dopo aver visto come si crea un'immagine con BitBake all'interno di Poky, vediamo come fare lo stesso usando Toaster. Ci concentreremo sull'utilizzo più semplice di Toaster citando anche cos'altro può fare Toaster per conoscerne le capacità.

### **Cos'è Toaster?**

Toaster è un'interfaccia web per configurare ed eseguire le build. Comunica con BitBake e col sistema di build Poky per gestire e raccogliere informazioni su build, pacchetti e immagini.

Ci sono due modi per utilizzare Toaster:

- **Locally:** Si può eseguire Toaster come istanza locale. Questo è adatto per lo sviluppo per un utente singolo, fornendo un'interfaccia grafica a riga di comando di BitBake e alcune informazioni sulla build.
- **Hosted:** Adatto per più utenti. Quando Toaster è configurato come istanza su host, i suoi componenti possono essere distribuiti su più macchine in modo che le build degli utenti vengano eseguite sui server di build Toaster.

In questo capitolo utilizzeremo Toaster come istanza locale. Per usarlo come istanza su host, consultare il sito web: <http://www.yoctoproject.org/docs/current/toaster-manual/toaster-manual.html>.

Si tenga presente che ogni servizio su host richiede una certa attenzione con la sua sicurezza. Riflettere prima di utilizzare un'istanza in hosting.

### **Installazione di Toaster**

Toaster è scritto col framework Python Django, quindi il modo più semplice per installarlo è usare l'utilità `pip` di Python. È stato già installato durante la configurazione della macchina host nel [Capitolo 2, Creazione del Sistema Poky-Based](#), quindi si può installare il resto richiesto da Toaster nella directory di Poky:

```
$ pip3 install --user -r bitbake/toaster-requirements.txt
```

### **Avvio di Toaster**

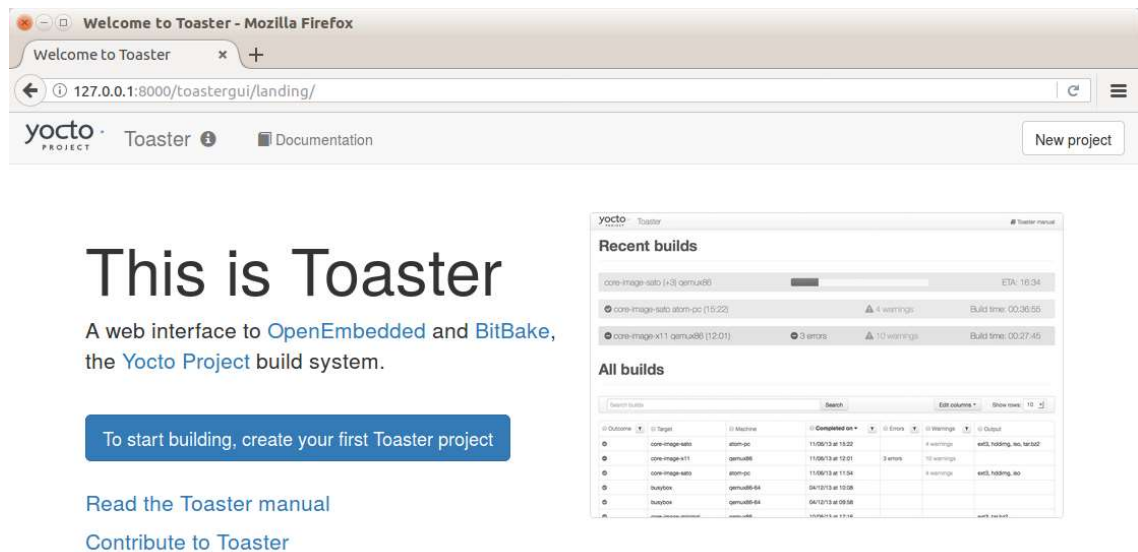
Una volta installati i requisiti di Toaster, si è pronti per avviarne il server. Per farlo, si deve andare nella directory di Poky e lanciare i seguenti comandi:

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

Per accedere all'interfaccia web di Toaster, ci si connette col browser all'indirizzo `http://127.0.0.1:8000`.

*Per default, Toaster si avvia sulla porta 8000. Il parametro `webport` consente di utilizzare una porta diversa, per esempio, `$ source toaster start webport=8400`.*

Apparirà la pagina iniziale di Toaster:

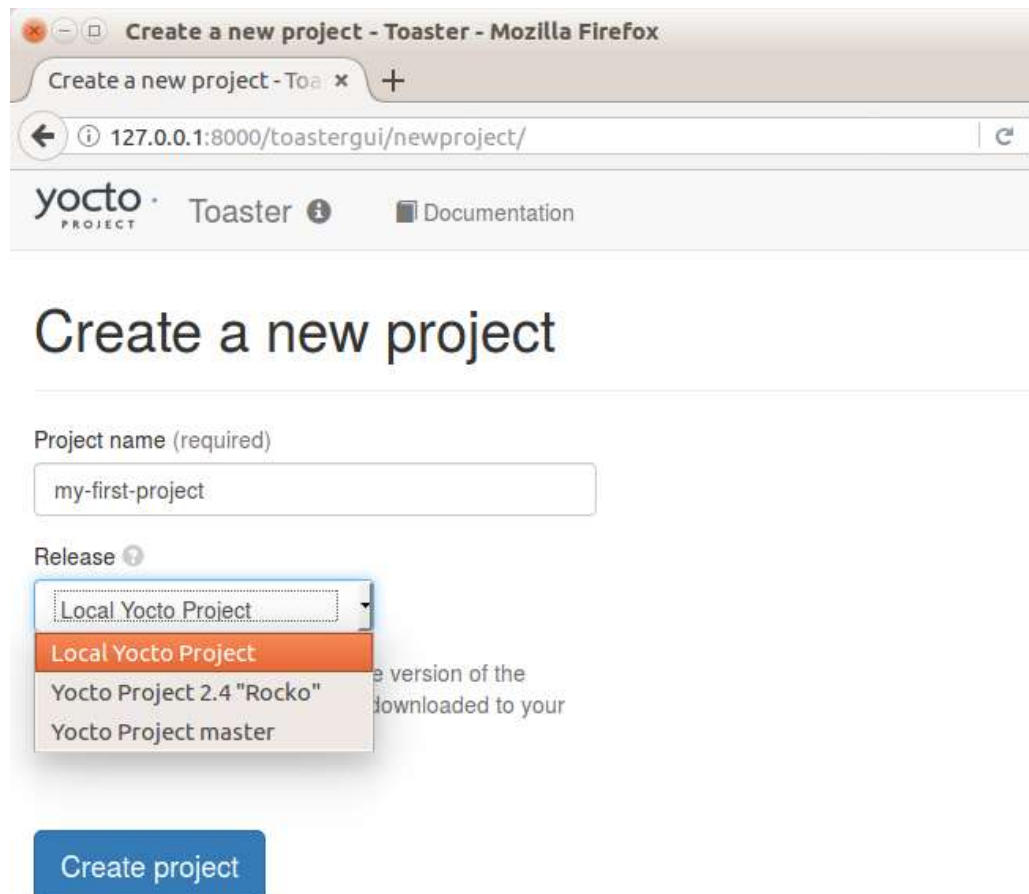


## Build di un'immagine per QEMU

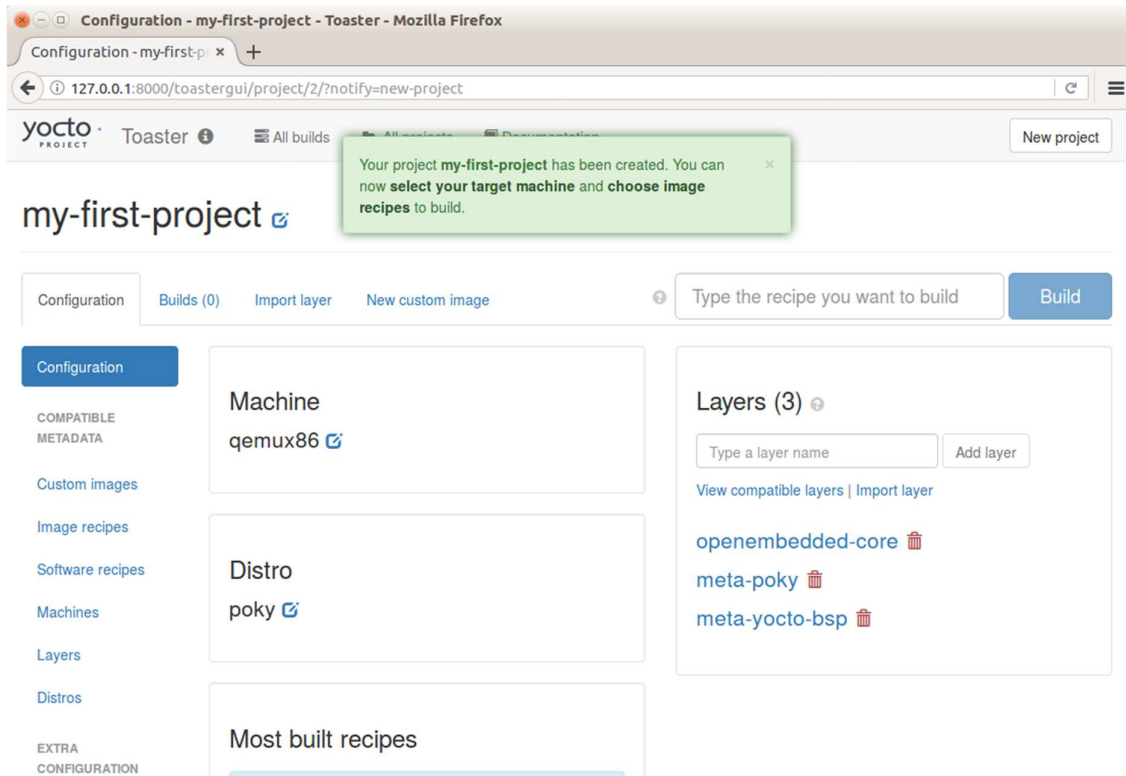
Seguendo gli stessi passaggi del [Capitolo 2, Creazione del Sistema Poky-Based](#), creeremo un'immagine per l'emulazione QEMU ARM.

Il primo passo consiste nel creare il primo progetto, che è una raccolta di configurazioni e build già eseguite.

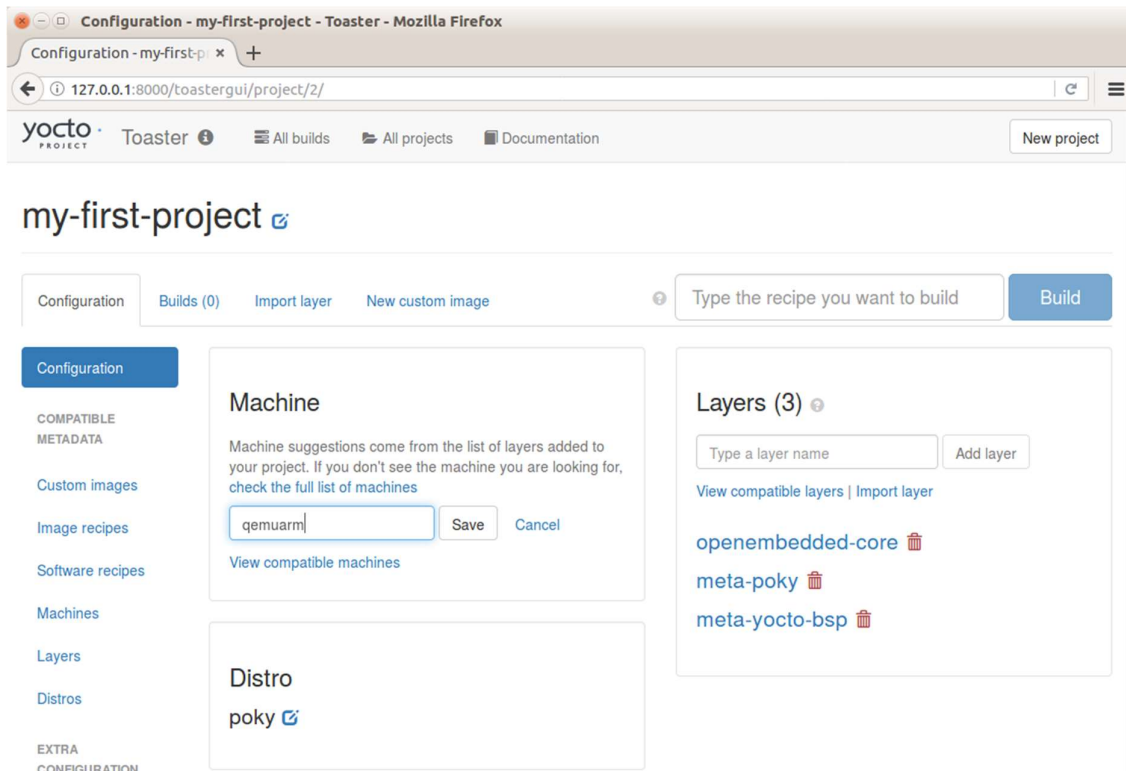
Non avendo un progetto, se ne deve avviare uno. Si crea un Project name e si sceglie il target del rilascio, come mostra la seguente schermata:



Dopo aver creato `my-first-project`, si può vedere la schermata principale del progetto, come in quella seguente:

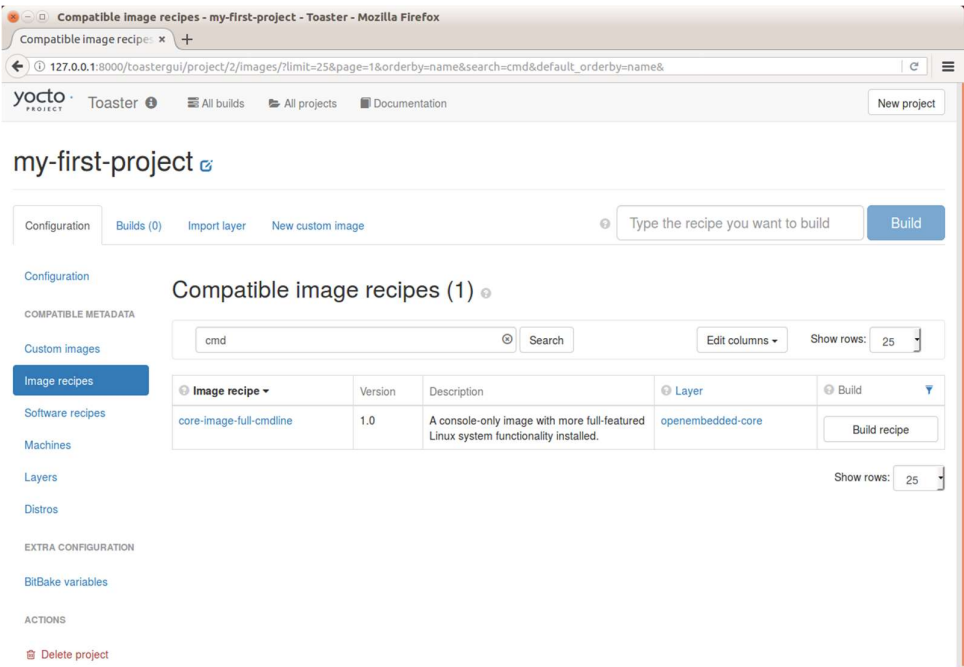


Nella scheda Configuration, si va su Machine e si modifica in `qemuarm`:

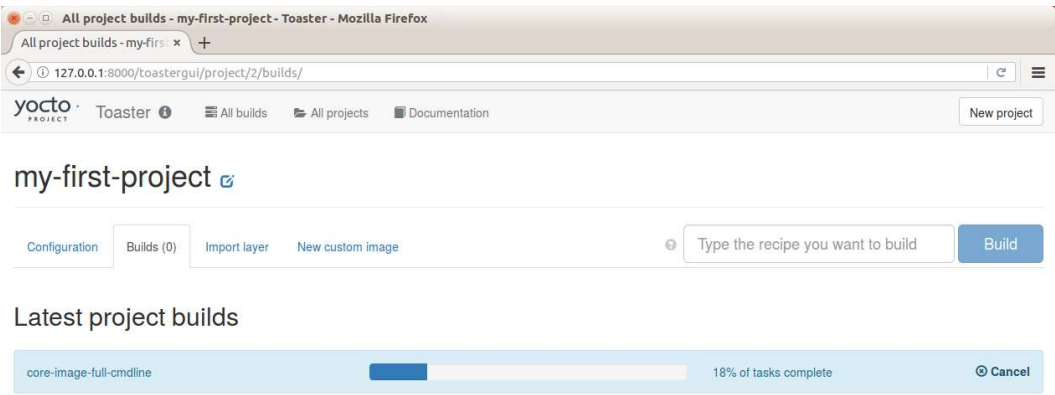


Poi si clicca sulla scheda Image Recipes e si sceglie l'immagine che si vuol 'buildare'. Il questo esempio, così come nel [Capitolo 2, Creazione del Sistema Poky-Based](#), si esegue la build di `core-image-full-cmdline`:

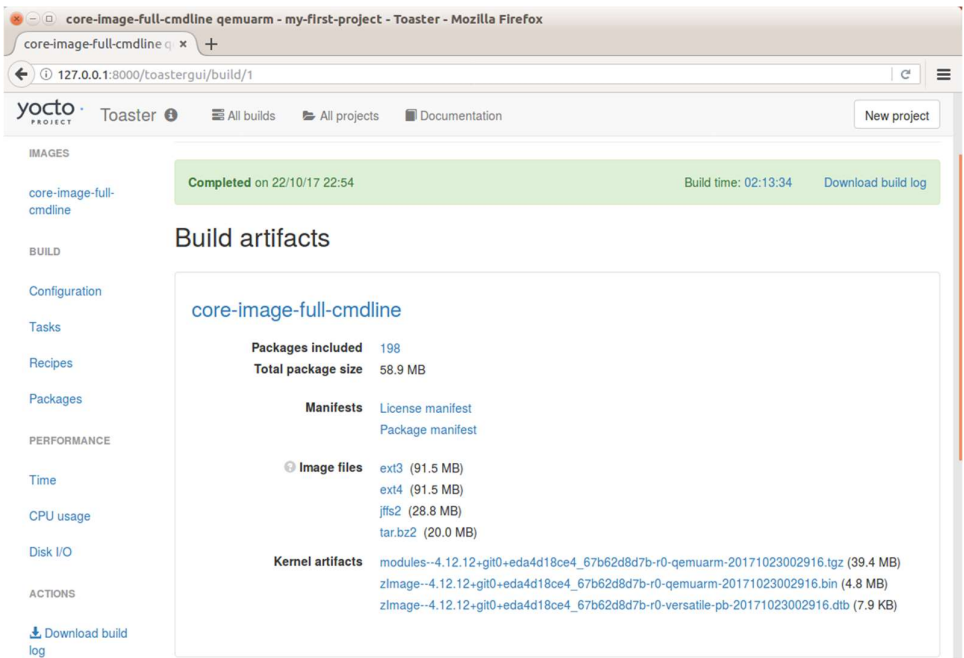




Lo screenshot seguente mostra il processo di build:



La build richiede del tempo, ma successivamente possiamo vedere l'immagine finale insieme ad alcune statistiche:



Si può anche controllare il set di file generati, come mostrato nella seguente schermata:

The screenshot shows the Yocto Project Toaster web interface in a Mozilla Firefox browser. The address bar shows the URL: 127.0.0.1:8000/toastergui/build/1/target/6/dirinfo. The page title is "Directory structure - core-image-full-cmdline qemuarm - my-first-project - Toaster - Mozilla Firefox". The interface includes a navigation bar with "All builds", "All projects", and "Documentation" links. The main content area displays the "core-image-full-cmdline" build summary. On the left, there is a sidebar with "BUILD" and "PERFORMANCE" sections. The "BUILD" section includes "Configuration", "Tasks", "Recipes", "Packages", "Time", "CPU usage", and "Disk I/O". The "PERFORMANCE" section includes "Time", "CPU usage", and "Disk I/O". The "Packages" section is selected, showing a table of packages included in the build. The table has columns for "Directory / File", "Symbolic link to", "Source package", "Size", "Permissions", "Owner", and "Group". The table lists several directories and files, all with a size of 4.0 KB and permissions of drwxr-xr-x, owned by root.

Directory / File	Symbolic link to	Source package	Size	Permissions	Owner	Group
bin			4.0 KB	drwxr-xr-x	root	root
boot			4.0 KB	drwxr-xr-x	root	root
dev			4.0 KB	drwxr-xr-x	root	root
etc			4.0 KB	drwxr-xr-x	root	root
home			4.0 KB	drwxr-xr-x	root	root
lib			4.0 KB	drwxr-xr-x	root	root
media			4.0 KB	drwxr-xr-x	root	root
mnt			4.0 KB	drwxr-xr-x	root	root
proc			4.0 KB	drwxr-xr-x	root	root
run			4.0 KB	drwxr-xr-x	root	root

Toaster è un tool potente. È utilizzabile sia su una macchina di sviluppo locale che su un server condiviso per ottenere una rappresentazione grafica della build. La maggior parte delle attività eseguite da BitBake si possono eseguire anche utilizzando Toaster.

## Sommario

In questo capitolo è stato presentato Toaster e le sue caratteristiche di base. È stato eseguito il processo di installazione e configurazione di Toaster ed è stata creata e ispezionata un'immagine.

Nel prossimo capitolo verranno introdotti alcuni importanti concetti di BitBake e concetti essenziali per comprendere appieno lo Yocto Project nel suo insieme. Si userà BitBake a riga di comando per il resto del libro per avere una visione dall'interno di tutti i concetti.

## 4. Capire il Tool BitBake

Inizieremo ora il viaggio per analizzare come funziona il motore dello Yocto Project dietro le quinte. Come in ogni viaggio, la comunicazione è fondamentale, c'è quindi bisogno di apprendere il linguaggio utilizzato dai tool dello Yocto Project per sfruttarli al massimo.

Nei capitoli precedenti, è stato seguito il workflow [flusso di lavoro] standard dello Yocto Project per la creazione e l'emulazione delle immagini. In questo capitolo, esploreremo il concetto di metadati, vedremo come le ricette dipendono l'una dall'altra e come queste vengono utilizzate da Poky.

Oltre alle dipendenze, ci sono altri aspetti importanti per ogni ricetta. Dal download del codice sorgente alla generazione delle immagini, c'è un enorme elenco di task [attività], come la memorizzazione del codice sorgente nella directory utilizzata per la build; applicazione di patch, configurazione, compilazione, installazione e generazione dei pacchetti; e determinare come i pacchetti rientrano nelle immagini generate.

### **Capire BitBake**

Il task scheduler BitBake è partito come una fork di Portage, che è il sistema di gestione dei pacchetti utilizzato nella distribuzione Gentoo. I due progetti, però, sono molto divergenti a causa dei diversi obiettivi di utilizzo. Lo Yocto Project e l'OpenEmbedded Project sono gli utenti più noti e intensivi di BitBake, che rimane un progetto separato e indipendente con un proprio ciclo di sviluppo e una propria mailing list ([bitbake-devel@lists.openembedded.org](mailto:bitbake-devel@lists.openembedded.org)).

Come esposto nel [Capitolo 1, Introduzione allo Yocto Project](#), BitBake è un task scheduler che esegue l'analisi [parsing] di codice Python misto a script shell. Sulla base dei metadati, BitBake genera un gran numero di task [attività] che possono avere una complessa catena di dipendenze, BitBake quindi garantisce che tali dipendenze siano soddisfatte massimizzando l'uso delle risorse ed eseguendo quante più attività possibili in parallelo. BitBake può essere visto come un tool simile a GNU Make per certi aspetti.

In questo capitolo si esamineranno gli aspetti principali di BitBake. Tuttavia, per approfondimenti, si faccia riferimento a <https://www.yoctoproject.org/docs/current/bitbake-user-manual/bitbake-user-manual.html>.

### **Esplorazione dei metadati**

I metadati utilizzati da BitBake si possono classificare in tre aree principali:

- Configurazione (i file `.conf`)
- Classi (i file `.bbclass`)
- Ricette (i file `.bb` e `.bbappend`)

I file di configurazione definiscono il contenuto globale, utilizzato per fornire informazioni e configurare il funzionamento delle ricette. Un esempio comune di file di configurazione è il file "machine", che contiene un elenco di impostazioni che descrivono l'hardware.

Le classi sono utilizzate dall'intero sistema e possono essere ereditate dalle ricette in base alle proprie esigenze, o per default, e sono utilizzate per definire il comportamento del sistema e fornire i metodi di base. Ad esempio, `kernel.bbclass` astrae i task [attività] relativi alla creazione e al building del kernel Linux indipendentemente dalla versione e al dal fornitore.

*Le ricette e le classi sono scritte in un mix di codice Python e di scripting shell.*

Le classi e le ricette descrivono i task [attività] da eseguire e forniscono le informazioni necessarie per consentire a BitBake di generare la catena di task richiesta, nonché le relative informazioni sulle dipendenze. Il meccanismo dell'ereditarietà che consente a una ricetta di ereditare una o più classi, viene utilizzato per ridurre la duplicazione del codice, migliorare la precisione e semplificare la manutenzione. Un esempio di ricetta del kernel Linux è `linux-yocto_4.12.bb`, che eredita un insieme di classi, tra cui `kernel.bbclass`.

Gli aspetti più comunemente utilizzati da BitBake in tutti i tipi di metadati (`.conf`, `.bb` e `.bbclass`) sono illustrati nelle sezioni seguenti.

La grammatica e la sintassi dei metadati sono descritte in dettaglio nel [Capitolo 7](#), *Approfondimenti sui Metadati di BitBake*.

## Analisi dei metadati

Come accennato in precedenza, esistono tre gruppi di metadati: configurazione, classe e ricetta.

I primi metadati analizzati in BitBake sono i metadati di configurazione, identificati dall'estensione del file `.conf`. Questi metadati sono globali e quindi interessano tutte le ricette e i task.

BitBake cerca prima nella directory di lavoro corrente il file di configurazione `build/conf/bblayers.conf`, che dovrebbe contenere una variabile `BBLAYERS` che è un elenco delimitato da spazi di directory di layer. Per ciascuna directory in questo elenco, viene cercato e analizzato un file `conf/layer.conf` per aggiungere le ricette, le classi e le configurazioni contenute in quel particolare layer. `LAYERDIR` è valorizzata con la directory in cui è stato trovato il layer durante questo processo e aggiunge anche questo layer all'elenco `BBPATH`, che punta all'insieme di directory con ricette, classi e configurazioni.

*L'ordine dei layer elencati nella variabile `BBLAYERS` viene seguito da BitBake durante il parsing dei metadati. Se il layer dev'essere analizzato prima, lo si deve elencare nella giusta posizione in `BBLAYERS`.*

Dopo aver esaminato tutti i layer in uso, BitBake inizia ad analizzare i metadati. Prima carica `meta/conf/bitbake.conf` da uno dei path inclusi nell'elenco di `BBPATH`. Similmente a molti altri file di configurazione BitBake, il file `meta/conf/bitbake.conf` utilizza le direttive `include` per inserire altri metadati, come quelli specifici dell'architettura, i file di configurazione della macchina e il file `build/conf/local.conf`. Un'importante restrizione dei file di configurazione BitBake (`.conf`) è che sono consentite solo definizioni di variabili e direttive `include`.

Le classi di BitBake (`.bbclass`), come accennato, costituiscono un meccanismo rudimentale di ereditarietà. Vengono analizzate quando si incontra una direttiva `inherit` e si trovano in `classes/`, relativamente alle directory in `BBPATH`.

Una ricetta BitBake (`.bb`) è un'unità logica di task [attività] da eseguire. Normalmente, questo è un pacchetto creato per obbedire alle dipendenze tra le ricette. I file stessi si trovano tramite la variabile `BBFILES`, che è valorizzata con un elenco separato da spazi dei file `.bb` e gestisce i caratteri jolly.

## Dipendenze

Per rispettarle, le ricette devono dichiarare quali dipendenze devono avere disponibili durante il processo di compilazione. BitBake soddisfa le dipendenze durante la build (build-time) prima di iniziare a la build della ricetta. Questo è più facile da capire se si pensa ad un'applicazione che usa una libreria. La build di tale libreria e i suoi header devono essere disponibili all'uso prima della stessa build dell'applicazione. La variabile

**DEPENDS** informa BitBake sulle dipendenze al build-time e ci dev'essere il listato di un'altra ricetta.

Quando un'applicazione dipende da qualcosa da eseguire, viene detta dipendenza a runtime. Ciò è solito per i dati condivisi tra le applicazioni (ad esempio le icone) che vengono utilizzati solo durante l'esecuzione dell'applicazione o quando un'applicazione ne chiama un'altra durante l'esecuzione che non viene utilizzata durante il processo di compilazione. Le dipendenze a runtime possono essere espresse utilizzando la variabile **RDEPENDS** in una ricetta. Tuttavia, poiché hanno lo scopo di esprimere i requisiti a runtime, qui si devono elencare i nomi dei pacchetti. Nel [Capitolo 6, Il supporto al Packaging](#), viene approfondito come avviene il packaging.

Conoscendo la catena delle dipendenze delle ricette, BitBake le può ordinare tutte per la build in un ordine fattibile. Ciò consente a BitBake di organizzare i task nei seguenti modi:

- Le ricette che non hanno una relazione di dipendenza vengono "buildate" in parallelo
- Le ricette dipendenti vengono "buildate" in sequenza, ordinate nel modo in cui le dipendenze sono soddisfatte.

Ogni ricetta inclusa nelle dipendenze a runtime viene inserita nell'elenco di build. Sembra ovvio, ma anche se non vengono utilizzate durante la compilazione, vengono incluse perché devono essere pronte per l'uso in modo che i pacchetti binari risultanti siano installabili.

## Input e output delle ricette

La relazione di dipendenza tra le ricette è essenziale per BitBake e Poky. È definita all'interno di ogni ricetta, con una variabile che descrive da cosa essa dipende (**DEPENDS**) e cosa fornisce una ricetta al sistema (**PROVIDES**). Queste due variabili insieme creano il grafico utilizzato da BitBake durante la risoluzione delle dipendenze.

Quindi, se una ricetta chiamata `foo_1.0.bb` dipende da `bar`, BitBake elenca tutte le ricette che forniscono `bar`. La dipendenza da `bar` può essere soddisfatta da:

- Una ricetta con il formato `bar_<version>.bb`, perché ogni ricetta fornisce se stessa
- Una ricetta in cui la variabile **PROVIDES** include la `bar`

Una dipendenza può essere soddisfatta da più ricette (ad esempio, se due o più ricette hanno **PROVIDES** `+= "bar"`). In questo caso, si deve informare BitBake su quale specifico provider utilizzare.

Il provider `virtual/kernel` è un chiaro esempio di come viene usato questo meccanismo. Il namespace `virtual/` è la convenzione adottata quando c'è un insieme di provider comunemente sovrascritti (overridden).

Tutte le ricette che richiedono la build del kernel possono aggiungere `virtual/kernel` all'elenco delle dipendenze (**DEPENDS**) e BitBake la soddisferà. Quando ci sono più ricette con un provider alternativo, se ne deve scegliere uno da utilizzare, ad esempio `PREFERRED_PROVIDER_virtual/kernel = "linux-mymachine"`.

Il provider `virtual/kernel` è impostato normalmente nel file di definizione della macchina, poiché può variare da una macchina all'altra. Vedremo come creare un file di definizione della macchina nel [Capitolo 11, Creazione di Layer Personalizzati](#).

*Quando BitBake non è in grado di soddisfare una dipendenza a causa di un provider mancante, viene generato un errore.*

Nello stesso sistema, due ricette non possono utilizzare provider diversi per la stessa dipendenza. Quando BitBake ha due provider con versioni diverse, utilizza la versione più

alta per default. Si può forzare BitBake a utilizzare una versione diversa utilizzando `PREFERRED_VERSION`. Questo avviene normalmente nei BSP, come i bootloader, dove i fornitori possono utilizzare versioni specifiche di una scheda.

Quando si ha uno sviluppo o una versione inaffidabile di una ricetta e non si vuole che venga utilizzata per default, si può usare `DEFAULT_PREFERENCE = "-1"` nella ricetta. Quindi, anche se la versione è superiore, non viene presa senza che sia stata impostata in modo esplicito (usando `PREFERRED_VERSION`).

## Prelievo del codice sorgente

Quando il codice sorgente di Poky viene scaricato, ciò che viene effettivamente copiato sono i metadati e BitBake. Il codice sorgente aggiuntivo viene prelevato su richiesta. Una delle principali funzionalità di BitBake è il prelievo del codice sorgente.

Questo supporto è stato progettato per essere il più modulare e flessibile possibile. Ogni sistema basato su Linux include il kernel Linux e molte altre utilità che formano il filesystem di root, come OpenSSH o un kernel Linux.

Il codice sorgente di OpenSSH è disponibile sul suo sito Web di upstream come file `tar.gz` su un server HTTP, mentre la versione del kernel Linux è solitamente su un repository Git e questi due diversi sorgenti possono essere facilmente prelevati da BitBake.

BitBake supporta molti tipi di fetcher che consentono il recupero di file tarball e una serie di altri protocolli, come Git, Subversion, Bazaar, OSC, HTTP, HTTPS, FTP, CVS, Mercurial, Perforce e SSH.

Il meccanismo utilizzato da BitBake per recuperare il codice sorgente è chiamato internamente come i "fetcher backend" che sono configurabili per allineare i requisiti dell'utente e per ottimizzare il fetching del codice sorgente.

## Download di file remoti

BitBake supporta diversi metodi per il download di file remoti. I più comunemente usati sono `http://`, `https://` e `git://`. Non tratteremo i dettagli interni di come BitBake gestisce i download di file remoti, ci concentreremo invece sui suoi effetti visibili.

Quando BitBake esegue il task `do_fetch` in una ricetta, controlla il contenuto di `SRC_URI`. Ad esempio, nella ricetta `libmpc` (disponibile su `meta/recipes-support/libmpc/libmpc_1.0.3.bb`), le variabili elaborate sono le seguenti:

```
SRC_URI = "http://www.multiprecision.org/mpc/download/mpc-${PV}.tar.gz"

SRC_URI[md5sum] = "d6a1d5f8ddea3abd2cc3e98f58352d26"

SRC_URI[sha256sum] = "617decc6ea09889fb08ede330917a00b16809b8db88c29c31bfbb49cbf88ecc3"
```

BitBake espande la variabile `PV` con la versione del pacchetto (1.0.3 in questo esempio), scarica il file da `http://www.multiprecision.org/mpc/download/mpc-1.0.3.tar.gz` e poi lo salva nella directory di download, definita dalla variabile `DL_DIR`.

Al termine del download, BitBake confronta i valori `md5sum` e `sha256sum` del file scaricato con i valori della ricetta e, se entrambi i valori corrispondono, crea un file `${DL_DIR}/mpc-1.0.3.tar.gz.done` per contrassegnare il file come scaricato e verificato correttamente. In questo modo, quando BitBake cerca il file la prossima volta, sa che può essere riutilizzato in sicurezza e salta la verifica del checksum del file. Questo processo avviene per ogni file remoto scaricato da BitBake.

Per default, la variabile `DL_DIR` punta a `build/downloads`. Lo si può sovrascrivere utilizzando il file `build/conf/local.conf` in questo modo: `DL_DIR = "/my/download-cache"`. Ciò semplifica la condivisione della stessa cache di download tra le diverse directory di build, risparmiando così tempo e occupazione di banda.

## I repository Git

Uno dei sistemi di gestione del controllo del codice sorgente più comunemente utilizzati è Git. BitBake ha un solido supporto per Git e il suo backend viene utilizzato si esegue il task `do_fetch` e si trova un URL `git://` nella variabile `SRC_URI`.

Per default il backend Git di BitBake gestisce i repository clonandoli in `${DL_DIR}/git2/<git URL>`. Per esempio, la seguente è uno stralcio dalla ricetta `linux-firmware_git.bb` che si trova in `meta/recipes-kernel/linux-firmware/linux-firmware_git.bb` all'interno di Poky:

```
SRCREV = "a61ac5cf8374edbf692d12f805a1b194f7fead2"
...
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git"
```

Qui, il repository `linux-firmware.git` viene clonato in `${DL_DIR}/git2/git.kernel.org.pub.scm.linux.kernel.git.firmware.linux-firmware.git`.

Questo nome di directory viene utilizzato per evitare conflitti tra altri possibili repository Git con lo stesso nome di progetto. La variabile `SRCREV` viene utilizzata dal task `do_fetch` per avere la revisione Git richiesta e forza un aggiornamento in caso contrario; viene utilizzato dal task `do_unpack` per impostare la directory di lavoro nella revisione del codice sorgente richiesta.

Quando la variabile `SRCREV` punta a un hash non disponibile nel branch [ramo] principale, è necessario utilizzare il parametro `branch=<branch name>` come segue: `SRC_URI = "git://myserver/myrepo.git;branch=mybranch"`. Nei casi in cui l'hash utilizzato punta a un tag che non è disponibile in un branch, dobbiamo usare l'opzione `nobranch=1` nel modo seguente: `SRC_URI = "git://myserver/myrepo.git;nobranch=1"`.

*Il file remoto e il repository Git sono i 'fetch' del backend di BitBake più comunemente usati. Gli altri sistemi di supporto per la gestione del codice sorgente variano nelle loro implementazioni, ma le idee e i concetti generali sono gli stessi.*

## Ottimizzazione del download dei sorgenti

Per migliorare la robustezza del download dei sorgenti, Poky fornisce un meccanismo di "mirror" configurabile per eseguire le seguenti operazioni:

- Fornire un server preferito centralizzato per il download
- Fornire server di riserva

Per gestire questo robusto meccanismo di download, BitBake segue alcuni passaggi. Durante il build, il primo passaggio di BitBake consiste nel cercare il codice sorgente all'interno della directory di download locale (specificata da `DL_DIR`). Se non lo trova, ritenta nelle posizioni definite dalla variabile `PREMIRRORS`. Se ancora non lo trova, ricerca nelle posizioni della variabile `MIRRORS`.

La sezione seguente illustra le variabili `PREMIRRORS` e `MIRRORS`:

- **PREMIRRORS:** Il secondo passaggio del meccanismo di download eseguito da BitBake è controllato dalla variabile `PREMIRRORS`. Per esempio, la distribuzione Poky lo imposta come segue:

```
PREMIRRORS ??= "\
bzip://.*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
cvs://.*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
git://.*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
git://.*.* *http://downloads.yoctoproject.org/mirror/sources/ \n \

hg://.*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
osc://.*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
```

```
p4://.*/*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
svn://.*/*.* http://downloads.yoctoproject.org/mirror/sources/ \n"
```

- Il codice precedente viene anteposto nella variabile **PREMIRRORS** per modificare e istruire il sistema di compilazione per intercettare qualsiasi richiesta Git, FTP, HTTP e HTTPS e le reindirizza al server mirror **http://www.yoctoproject.org/sources/**.
- Nel caso in cui il componente desiderato non sia disponibile nel mirror, BitBake ricorre alla variabile **MIRRORS**.
- **MIRRORS**: La variabile **MIRRORS** fornisce una serie di indirizzi di URL alternativi per alcuni server in cui è possibile trovare il componente. BitBake prova uno dopo l'altro e, se tutti i mirror disponibili falliscono, viene visualizzato un errore:

```
MIRRORS += "\
ftp://.*/*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
http://.*/*.* http://downloads.yoctoproject.org/mirror/sources/ \n \
https://.*/*.* http://downloads.yoctoproject.org/mirror/sources/ \n"
```

Si può sfruttare questo meccanismo di download per risparmiare tempo fornendo una directory locale o un server di rete locale aggiungendo il seguente codice a **build/conf/local.conf**:

```
SOURCE_MIRROR_URL ?= "file:///home/you/your-download-dir/"
INHERIT += "own-mirrors"
```

Qui, **SOURCE\_MIRROR\_URL** può far puntare a una directory locale o a qualsiasi URL di un server il backend fetcher supportato.

Se l'obiettivo è avere una cache per il download condivisa, è consigliabile abilitare la generazione del tarball per i backend SCM (ad esempio Git) nella cartella di download con **BB\_GENERATE\_MIRROR\_TARBALLS = "1"** in **build/conf/local.conf**.

## Disabilitare l'accesso alla rete

Talvolta, è necessario disabilitare Internet durante il processo di compilazione. Ci sono diverse ragioni valide per farlo, tra cui:

- **Policy**: La nostra azienda non consente l'inclusione di sorgenti esterne in un prodotto senza un'adeguata convalida e revisione legale.
- **Costo delle rete**: Quando si è in viaggio utilizzando la banda larga mobile, il costo dei dati potrebbe essere troppo elevato, data la mole.
- **Divisione del Download dal build**: Questa configurazione è solitamente utilizzata in ambienti di integrazione costante in modo che ci sia un job per il download di tutto il codice sorgente mentre gli addetti alla build hanno l'accesso a Internet disabilitato. Questo per non scaricare duplicati e che tutto il codice sorgente sia memorizzato nella cache.
- **Mancanza di connessione alla rete**: Talvolta non si ha accesso a una rete

Per disabilitare la connessione, si deve aggiungere il seguente codice nel file **build/conf/local.conf** file:

```
BB_NO_NETWORK = "1"
```

## Il ruolo di BitBake

BitBake utilizza le unità di esecuzione, che sono essenzialmente un gruppi di istruzioni eseguite in sequenza. Queste unità sono note come **task** [attività]. Ci sono molti task pianificati, eseguiti e controllati da BitBake durante la build di ciascuna ricetta, fornite dalle classi per formare il framework usato per la build di una ricetta. È importante comprenderne alcuni poiché spesso usati, si estensi, e implementati o sostituiti durante la scrittura di una ricetta.

Eseguire il seguente comando:

```
$ bitbake <recipe>
```



BitBake esegue una serie di task [attività] schedulate [pianificate]. Quando si vuole eseguire una specifica attività, possiamo utilizzare il seguente comando:

```
$ bitbake <recipe> -c <task>
```

Per elencare i task definiti in una ricetta, possiamo usare il seguente comando:

```
$ bitbake <recipe> -c listtasks
```

Si descriverà brevemente ognuno:

- **do\_fetch:** Il primo passo nel build di una ricetta consiste nel recuperare i sorgenti richiesti. Questo viene fatto utilizzando la funzione di 'fetch' discussa in precedenza in questo capitolo. È importante sottolineare che il prelievo dei sorgenti o di un file non significa che si tratti di prelevare da remoto. In effetti, ogni file necessario alla build della ricetta deve essere recuperato in modo che sia reso disponibile nella directory `WORKDIR`. Si approfondirà sulla directory di build e sui suoi contenuti nel [Capitolo 5, Dettagli sulla Directory Temporanea di Build](#). Tutto il contenuto scaricato viene archiviato nella cartella di download (la variabile `DL_DIR`), poi tutto il codice sorgente esterno viene memorizzato nella cache per evitare di scaricarlo ogni volta che è richiesto.
- **do\_unpack:** Il naturale task successivo, dopo il `do_fetch` è `do_unpack`. È responsabile della decompressione del codice sorgente o del controllo della revisione o del branch richiesti nel caso in cui la si utilizzi un sistema SCM.
- **do\_patch:** Una volta che il codice sorgente è stato decompresso correttamente, BitBake avvia il processo di adattamento. Questo viene fatto dal `do_patch`. Si presume che ogni file recuperato da `do_fetch`, con estensione `.patch`, sia una patch da applicare. Questo task applica l'elenco delle patch necessarie.
- **do\_configure, do\_compile e do\_install:** I task `do_configure`, `do_compile` e `do_install` vengono eseguiti in questo ordine. Alcune ricette possono omettere uno o più task. È importante notare che le variabili di ambiente definite nei task sono diverse per ogni task. I task variano molto da una ricetta all'altra. Poky fornisce una ricca raccolta di task predefinite nelle classi, che dovrebbero essere utilizzate quanto più possibile. Ad esempio, quando la classe `autotools` viene ereditata da una ricetta, fornisce un'implementazione nota dei task `do_configure`, `do_compile` e `do_install`.
- **do\_package:** Il task `do_package` suddivide i file installati dalla ricetta, in componenti logici come simboli di debug, documentazione e librerie. Il task `do_package` suddivide e impacchetta correttamente i file. Gli approfondimenti sul packaging stanno nel [Capitolo 6, Il Supporto al Packaging](#).

## Estensione dei task

Quando il contenuto del task non soddisfa i requisiti, lo si sostituisce (fornendo una nuova implementazione) o vi si aggiunge qualcosa. Come si vedrà più ampiamente sulla sintassi dei metadati BitBake nel [Capitolo 7, Approfondimenti sui Metadati di BitBake](#), si possono usare gli operatori `_append` e `_prepend` per estendere un task con del contenuto extra. Il nuovo contenuto viene concatenato a quello del task originale. Ad esempio, per estendere un task `do_install`, si può utilizzare il codice seguente:

```
do_install_append() {
# Do my commands
}
```

I meccanismi utilizzabili per estendere le ricette esistenti sono trattati nel [Capitolo 12, Personalizzazione delle Ricette Esistenti](#).

## Generazione di un'immagine del filesystem di root

Uno degli usi più comuni di Poky è la generazione di immagini di `rootfs`. L'immagine di `rootfs` dovrebbe essere vista come un filesystem di root pronto all'uso per un target. L'immagine può essere composta da uno o più filesystem e può includere altri artefatti che

saranno disponibili durante la sua generazione, come il kernel Linux, l'albero dei dispositivi e i binari del bootloader. Il processo di generazione dell'immagine è composto da diversi passaggi e i suoi usi più comuni sono i seguenti:

1. Generare la directory di `rootfs`.
2. Creare i file richiesti.
3. Racchiudere il filesystem finale in base ai requisiti specifici (potrebbe essere un file del disco con diverse partizioni e contenuti).
4. Infine, la compressione, se è possibile.

Tutti questi passaggi vengono eseguiti dai sub-task di `do_rootfs`.

`rootfs` è fondamentalmente una directory con i pacchetti installati desiderati, subito dopo l'applicazione delle modifiche (la generazione dei pacchetti è trattata nel [Capitolo 6, // Supporto al Packaging](#)). Le modifiche apportano piccole modifiche ai contenuti di `rootfs`; ad esempio, durante la creazione di un'immagine di sviluppo, `rootfs` viene modificato per consentire l'accesso come root senza password..

L'elenco dei pacchetti da installare in `rootfs` è definito dall'unione dei pacchetti elencati da `IMAGE_INSTALL` e dei pacchetti inclusi da `IMAGE_FEATURES`; la personalizzazione dell'immagine è descritta in dettaglio nel [Capitolo 11, Creazione di Layer Personalizzati](#). Ogni funzionalità dell'immagine può includere pacchetti extra per l'installazione, ad esempio `dev-pkgs`, che installa le librerie di sviluppo e gli header di tutti i pacchetti da installare in `rootfs`.

L'elenco dei pacchetti da installare viene poi filtrato dalla variabile `PACKAGE_EXCLUDE`, che elenca i pacchetti che non devono essere installati. I pacchetti elencati in `PACKAGE_EXCLUDE` vengono esclusi solo dall'elenco dei pacchetti da installare esplicitamente.

*I pacchetti elencati in `PACKAGE_EXCLUDE` vengono installati nel `rootfs` se sono necessari per soddisfare una dipendenza di runtime.*

Col set finale di pacchetti da installare, il task `do_rootfs` può avviare il processo di decompressione e configurazione di ogni pacchetto e delle sue dipendenze, nella directory `rootfs`. Questo viene fatto utilizzando un set di task secondari specifici del back-end di un pacchetto (DEB, IPK o RPM) poiché utilizza effettivamente il sistema di gestione dei pacchetti partendo da pacchetti locali per eseguire questo passaggio. Il prelievo [feed] dei pacchetti viene spiegato nel [Capitolo 6, // Supporto al Packaging](#).

Con il contenuto di `rootfs` decompresso, gli script di post installazione dei pacchetti indicati devono essere eseguiti per evitare di eseguirli durante il primo boot [avvio]. Potrebbe essere necessario eseguire alcuni script nel target per completare il lavoro e non ci sono problemi con questo, tranne quando si utilizza la funzionalità `read-only-rootfs` dell'immagine. Lo script di post-installazione e le altre sue varianti vengono trattati nel [Capitolo 6, // Supporto al Packaging](#).

*Tutti gli script di post-installazione devono essere eseguiti con una directory `rootfs` `read-only`.*

In seguito viene eseguita l'ottimizzazione di `rootfs`. Un processo di `prelink` ottimizza il link dinamico delle librerie shared (condivise) per ridurre il tempo di avvio degli eseguibili; il processo `mklibs` ottimizza la dimensione delle librerie rimuovendo i simboli non utilizzati. A questo punto, la directory è pronta per generare il filesystem. `IMAGE_FSTYPES` elenca il filesystem da generare, ad esempio `EXT4` o `UBIFS`.

Al completamento di `do_rootfs`, il file di immagine generato viene inserito in `build/tmp/deploy/image/<machine>/`. Il processo di creazione dell'immagine e i

possibili valori per `IMAGE_FEATURES` e `IMAGE_FSTYPES` vengono descritti nel [Capitolo 11](#), *Creazione di Layer Personalizzati*.

## Sommario

In questo capitolo abbiamo appreso il concetto di metadati, come le ricette dipendono l'una dall'altra e come Poky gestisce le dipendenze. Abbiamo avuto una visione più ampia dei task gestiti da BitBake per scaricare tutto il codice sorgente, utilizzato la build e per generare i pacchetti, e si è visto come tali pacchetti vengono inseriti nelle immagini.

Nel prossimo capitolo vedremo il contenuto della directory di build dopo la generazione dell'immagine, si vedrà come BitBake la usa nel processo di "baking" [cottura], incluso il contenuto della directory temporanea di build ed i file generati.

## 5. Dettagli sulla Directory Temporanea di Build

In questo capitolo si esamina il contenuto della directory di build temporanea dopo una generazione completa dell'immagine e si vedrà come BitBake la usa nel processo di baking. Vedremo come alcune di queste directory possono aiutare a comprendere quando le cose non funzionano come previsto, fornendo una preziosa fonte di informazioni.

### ***Dettagliare la build directory***

La build directory è una fonte centrale di informazioni e artefatti per ogni utente Poky. Le directory principali sono le seguenti:

- **conf**: Contiene i file di configurazione per controllare Poky e BitBake. Questa directory è stata usata per la prima volta nel [Capitolo 2, Creazione del Sistema Poky-Based](#). Contiene i file di configurazione come `build/conf/local.conf` e `build/conf/bblayers.conf`.
- **downloads**: Contiene tutti gli artefatti scaricati. Può essere vista come la cache di download ed è stata descritta in dettaglio nel [Capitolo 4, Capire il Tool BitBake](#).
- **sstate-cache**: Contiene gli snapshot dei dati impacchettati. È una cache utilizzata principalmente per accelerare il processo di build. Questa cartella è descritta in dettaglio nel [Capitolo 6, Il Supporto al Packaging](#).
- **tmp**: Questa è la directory di build temporanea e l'oggetto principale di questo capitolo.

### ***Costruzione della build directory***

Gli input e gli output di Poky sono già stati descritti in dettaglio ad un alto livello di astrazione nei capitoli precedenti. È noto che BitBake utilizza i metadati per generare diversi tipi di artefatti, comprese le immagini. Oltre agli artefatti generati, BitBake crea molti più contenuti durante questo processo, utilizzabili in diversi modi, a seconda degli obiettivi.

Durante il processo di compilazione, BitBake esegue diversi task e modifica continuamente la directory di build. Lo si comprende meglio seguendo il solito flusso di BitBake, come segue:

- **Fetching**: La prima azione realizzata da BitBake è quella di scaricare il codice sorgente. Questo passaggio può modificare la directory di build nel tentativo di utilizzare la copia del codice sorgente, o quando scarica nella cache, oppure esegue il download e memorizza nella directory `build/download`.
- **Preparazione del sorgente**: Dopo aver recuperato il codice sorgente, è necessario prepararlo per l'uso. Ciò può comportare, ad esempio, la decompressione di un tarball o eseguire un clone di una directory Git nella cache locale (dalla cache di download). Il codice sorgente viene preparato nella directory `build/tmp/work`. Quando è pronto, vengono applicate le modifiche richieste (ad esempio, l'applicazione delle patch necessarie).
- **Configurazione e build**: Col codice sorgente pronto per l'uso, inizia il processo di build. Implica la configurazione delle opzioni di build (ad esempio, `./configure`) e il build stesso (ad esempio, `make`).
- **Installazione**: Gli artefatti compilati vengono quindi installati (ad esempio, `make install`) in una directory appropriata sotto `build/tmp/work/<...>/image`.
- **Wrapping di sysroot**: Le librerie, gli header e altri file che devono essere condivisi per la cross-compilazione vengono copiati (e talvolta modificati) in

```
build/tmp/work/<...>/recipe-sysroot e
build/tmp/work/<...>/recipe-sysroot-native.
```

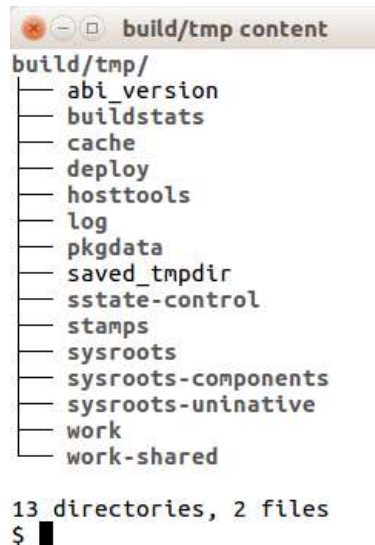
- Creazione dei pacchetti: I pacchetti vengono generati utilizzando il contenuto installato, potenzialmente suddividendo questo contenuto in più pacchetti, disponibili in diversi formati, ad esempio `.rpm` or `.ipk`.

Fino alla release dello Yocto Project 2.2 (Morty), si è utilizzata la directory `sysroot` globale per ciascuna macchina e architettura, ma dalla versione dello Yocto Project 2.3 (Pyro), c'è una directory `sysroot` per ciascuna ricetta, aumentando il determinismo del sistema di build.

## Esplorazione della directory di build temporanea

È fondamentale comprendere la directory di build temporanea (`build/tmp`). La directory di build temporanea viene creata subito dopo l'avvio della build ed è particolarmente importante per identificare il motivo per cui qualcosa non si è comportato come previsto.

Il contenuto della directory `build/tmp` è mostrato nella figura seguente:



```
build/tmp/
├── abi_version
├── buildstats
├── cache
├── deploy
├── hosttools
├── log
├── pkgdata
├── saved_tmpdir
├── sstate-control
├── stamps
├── sysroots
├── sysroots-components
├── sysroots-uninative
├── work
└── work-shared

13 directories, 2 files
$
```

Le directory più importanti che si trovano al suo interno sono le seguenti:

- **deploy**: Contiene i prodotti della build come immagini, pacchetti binari e gli SDK
- **sysroots**: Contiene le librerie shared [condivise], GLI HEADER e le utilità utilizzate nel processo di BUILD delle ricette
- **work**: Contiene il codice sorgente funzionante, la configurazione di un task, i log delle esecuzioni e il contenuto dei pacchetti generati

## Capire la directory di lavoro

La directory `build/tmp/work` è organizzata per architetture. Ad esempio, lavorando con la macchina `qemuarm`, si hanno le seguenti quattro directory:

- `all-poky-linux`
- `armv5te-poky-linux-gnueabi`
- `qemuarm-poky-linux-gnueabi`
- `x86_64-linux`

Queste directory e il loro contenuto dipendono dall'architettura e dalla macchina. Non lo si deve prendere come un elenco finale, solo come un'indicazione. La directory `x86_64-linux` viene usata per il build del contenuto della `sysroot` dell'host, descritto in dettaglio nella prossima sezione. La directory `all-poky-linux` contiene le directory di build funzionanti per i pacchetti indipendenti dall'architettura. Questa struttura frammentata è

necessaria per consentire il build di più macchine e architetture all'interno di una directory di build, senza che entrino in conflitto.

La macchina target che si utilizza è la `qemuarm`. Questa macchina è un'emulazione della ARM Versatile Platform Baseboard con l'emulazione della CPU ARM926EJ-S che supporta le istruzioni ARMv5TE. Poky considera `qemuarm` come un tipo di ARMv5TE perché alcune funzionalità hardware potrebbero non essere disponibili su un dispositivo o un altro, anche quando sono supportate dalla CPU. Il build delle ricette specifiche per la macchina avviene nella directory della macchina (`qemuarm-poky-linux-gnueabi` in questo caso) mentre il build dei pacchetti specifici per l'architettura sono nella directory specifica per l'architettura (in questo caso `armv5te-poky-linux-gnueabi`).

La directory `build/tmp/work` è utilissima quando si cercano comportamenti scorretti o errori di build. I suoi contenuti sono archiviati in sottodirectory seguendo questo schema:

```
<arch>/<recipe name>/<software version>
```

Alcune delle directory sotto questa struttura sono:

- **<sources>**: Questo è un estratto del codice sorgente del software da creare. Questa è la directory puntata dalla variabile `WORKDIR`.
- **image**: Questa contiene i file installati dalla ricetta (puntata dalla variabile `D`).
- **packages**: Il contenuto estratto dai pacchetti viene memorizzato qui.
- **packages-split**: Il contenuto dei pacchetti, estratto e suddiviso, viene memorizzato qui. Ha una sub-directory per ciascun pacchetto.
- **temp**: Contiene i log del codice e dell'esecuzione dei task di BitBake.

Le sottodirectory controllate più spesso si trovano nella directory `sysroot`, che contiene gli artefatti utilizzati durante la cross-compilazione come compilatori, utilità e librerie per l'host e il target; viene anche controllata la directory `build/tmp/work`, che contiene la directory build di lavoro. Queste directory forniscono informazioni preziose per il debug.

*Per ridurre l'uso del disco, si può automaticamente rimuovere la directory work dopo ogni ciclo di compilazione di una ricetta, aggiungendo `INHERIT += "rm_work"` nel file `build/conf/local.conf`.*

La struttura della directory `work` è la stessa per tutte le architetture. Per ogni ricetta viene creata una directory con il nome della ricetta stessa. Prendendo la directory di lavoro specifica della macchina e usando la ricetta `sysvinit-inittab` come esempio, vediamo questo:

```
sysvinit-inittab content inside build/tmp/work
work/qemuarm-poky-linux-gnueabi/sysvinit-inittab/2.88dsf-r10/
├── configure.sstate
├── deploy-rpms
├── image
├── inittab
├── license-destdir
├── package
├── packages-split
├── patches
├── pkgdata
├── pseudo
├── recipe-sysroot
├── recipe-sysroot-native
├── start_getty
├── sysroot-destdir
├── sysvinit-inittab.spec
└── temp

12 directories, 4 files
$
```

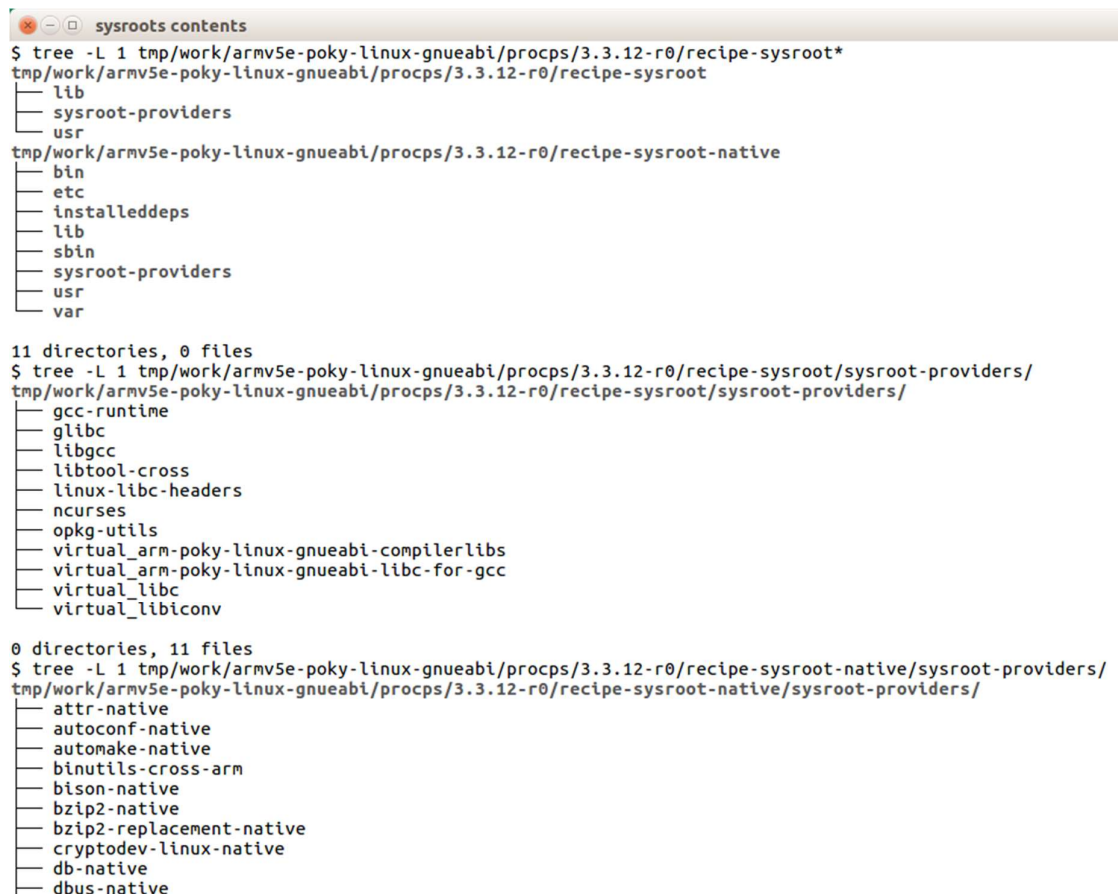
La ricetta `sysvinit-inittab` è un buon esempio, in quanto pur senza un codice oggetto specifico della macchina è specifica della macchina. Contiene il file `inittab` che definisce, tra le altre cose, i device della console seriale per generare il processo di login, e questo varia da macchina a macchina poiché la UART dipende dal layout schematico della macchina.

Le directory mostrate nella figura precedente che non sono descritte in dettaglio qui vengono utilizzate dal sistema di build. Non dovrebbe essere necessario gestirle, a meno che non si stia lavorando allo sviluppo di tool per il build.

La directory `work` è molto utile per il debug; lo tratteremo nel [Capitolo 9](#), *Debugging con lo Yocto Project*.

## Capire le directory sysroot

Tradizionalmente, la directory `sysroot` dello Yocto Project è stata condivisa tra tutte le ricette e l'environment del sistema di build, ma ciò presenta una serie di carenze poiché questo macro ambiente ha tutte le dipendenze di tutte le ricette 'buildate' in precedenza e tali librerie e utilità possono influenzare altre ricette. A partire dallo *Yocto Project 2.4 (Rocko)*, la struttura della `sysroot` è stata migliorata per utilizzare una `sysroot` specifica per la ricetta. Il contenuto delle directory `sysroot` è mostrato nella figura seguente:



```

$ tree -L 1 tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot*
tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot
├── lib
├── sysroot-providers
├── usr
└── tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot-native
    ├── bin
    ├── etc
    ├── installeddeps
    ├── lib
    ├── sbin
    ├── sysroot-providers
    ├── usr
    └── var

11 directories, 0 files
$ tree -L 1 tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot/sysroot-providers/
tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot/sysroot-providers/
├── gcc-runtime
├── glibc
├── libgcc
├── libtool-cross
├── linux-libc-headers
├── ncurses
├── opkg-utils
├── virtual_arm-poky-linux-gnueabi-compilerlibs
├── virtual_arm-poky-linux-gnueabi-libc-for-gcc
├── virtual_libc
└── virtual_libconv

0 directories, 11 files
$ tree -L 1 tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot-native/sysroot-providers/
tmp/work/armv5e-poky-linux-gnueabi/procps/3.3.12-r0/recipe-sysroot-native/sysroot-providers/
├── attr-native
├── autoconf-native
├── automake-native
├── binutils-cross-arm
├── bison-native
├── bzip2-native
├── bzip2-replacement-native
├── cryptodev-linux-native
├── db-native
└── dbus-native
  
```

Dopo il build della ricetta `procps`, versione 3.3.12, si ottengono due set di directory `sysroot`, come mostrato nella schermata precedente. Le directory sono `recipe-sysroot-native` e `recipe-sysroot`, e all'interno di ogni set di `sysroot` c'è una sottodirectory chiamata `sysroot-provides`. Questa directory contiene i pacchetti installati su ogni `sysroot`.

`recipe-sysroot-native` include le dipendenze di compilazione utilizzate nel sistema host durante il processo di build. È fondamentale per il processo di cross-compilazione perché comprende il compilatore, il linker, gli strumenti di script di build e altro, mentre

nella `recipe-sysroot` abbiamo le librerie e gli header utilizzati nel codice di target; nel nostro esempio è stato usato `qemuarm`.

Un errore comune da evitare quando si progetta una ricetta di libreria è non installare correttamente il suo contenuto (gli header, le librerie statiche e quelle shared) nella directory puntata alla variabile `D` in modo che Poky possa, il più delle volte, eseguire la corretta installazione dei file necessari in `sysroot`. Quando si vede un header mancante o un link non riuscito, si deve ricontrollare se i contenuti di `sysroot` (target e host) sono corretti.

## Sommario

In questo capitolo, è stato esplorato il contenuto della directory di build temporanea dopo una generazione completa dell'immagine e si è visto come BitBake la usa durante il processo di baking. Si è poi visto come utilizzare queste directory per il debug.

Nel prossimo capitolo, si capirà meglio come viene eseguito il packaging in Poky, come utilizzare i feed dei pacchetti e il servizio PR e come possono aiutare nella manutenzione del prodotto.



## 6. Il supporto al Packaging

Questo capitolo presenta i concetti chiave per comprendere gli aspetti di Poky e BitBake relativi al packaging. Si vedranno i formati dei pacchetti binari supportati, la cache di stato condivisa, i componenti per il controllo delle versioni dei pacchetti, come impostare e utilizzare i `feed dei pacchetti` binari per supportare il processo di sviluppo e altro ancora.

### **Utilizzo dei formati di package supportati**

I package [pacchetti] sono fondamentali per Poky in quanto consentono al sistema di build di produrre diversi tipi di artefatti, come immagini e toolchain.

Una ricetta può generare uno o più pacchetti come risultato dell'esecuzione di BitBake. D'altra parte, le immagini e le toolchain sono costituite da diversi pacchetti che vengono decompressi e configurati per raggiungere l'obiettivo prefissato. Il risultato generato viene racchiuso in modo tale da poter essere installato in una o più immagini oppure essere distribuito per un uso successivo.

### **Elenco dei formati di package supportati**

Attualmente, BitBake supporta quattro diversi formati di package:

- **RPM:** Il nome originale era `Red Hat Package Manager (RPM)`, ma ora noto come formato del pacchetto RPM dalla sua adozione da parte di diverse altre distribuzioni Linux, è utilizzato da popolari distribuzioni Linux come SuSE, OpenSuSE, Red Hat, Fedora e CentOS, tra gli altri.
- **DEB:** Il `Debian Package Manager` è usato da Debian e da molte altre distribuzioni basate su Debian; i più conosciuti tra loro sono Ubuntu Linux e Linux Mint.
- **IPK:** L'`Itsy Package Management System` era un sistema leggero di gestione dei pacchetti progettato per dispositivi embedded che assomigliava al formato di Debian. Il progetto iniziale è stato interrotto e molti dei sistemi di build embedded e delle distribuzioni che lo utilizzavano sono stati spostati sul `Opkg fork` creato da OpenMoko; è utilizzato anche dal progetto `OpenWRT`. Attualmente, OpenEmbedded-Core, e di conseguenza Poky, utilizzano il gestore Opkg per supportare il formato IPK.
- **TAR:** Questo deriva dall'archiviazione su nastro `.tar`, ed è un tipo di file tarball ampiamente utilizzato per raggruppare diversi file in un unico file.

### **Scelta del formato del pacchetto**

Il supporto per i formati viene fornito utilizzando un insieme di classi (`package_rpm`, `package_deb` e `package_ipk`). Si possono selezionare uno o più formati tramite la variabile `PACKAGE_CLASSES`, come mostrato nell'esempio seguente:

```
PACKAGE_CLASSES ?= "package_rpm package_deb package_ipk"
```

Questo può essere fatto, ad esempio, nel file `build/conf/local.conf` file. Utilizzando la variabile `PACKAGE_CLASSES`, si possono generare i pacchetti in uno o più formati.

*Le immagini vengono create dal primo formato di pacchetto trovato in `PACKAGE_CLASSES`.*

Per default, Poky utilizza il formato `RPM`, che usa il gestore di pacchetti `DNF`. Tuttavia, la scelta del formato dipende da diversi aspetti, tra cui le funzionalità specifiche del formato, l'utilizzo della memoria, delle risorse e così via. Un altro aspetto è l'abitudine; ad esempio, gli utenti di OpenEmbedded-Core spesso si sentono più a loro agio nell'usare `IPK` e `opkg` come gestore di pacchetti poiché è il default di OpenEmbedded-Core e offre un ingombro ridotto in termini di memoria e utilizzo delle risorse. D'altra parte, le persone abituate ai sistemi Debian potrebbero preferire i formati `APT` e `DEB` per i loro prodotti.

## Codice in esecuzione durante l'installazione del pacchetto

I pacchetti possono utilizzare gli script come parte del processo di installazione e rimozione. Gli script inclusi sono definiti come segue:

- **preinst**: viene eseguito prima che il pacchetto venga decompresso. I servizi devono essere interrotti durante l'esecuzione di **preinst** per consentire l'installazione o l'aggiornamento.
- **postinst**: Questo in genere completa qualsiasi configurazione richiesta del pacchetto dopo che è stato decompresso. Molti script **postinst** eseguono quindi qualsiasi comando necessario per avviare o riavviare un servizio una volta che un nuovo pacchetto è stato installato o aggiornato.
- **prerm**: Questo di solito arresta qualsiasi demone associato a un pacchetto. Viene eseguito prima della rimozione dei file associati al pacchetto.
- **postrm**: Questo comunemente modifica i collegamenti o altri file creati dal pacchetto.

Gli script dovrebbero essere eseguiti dopo che l'installazione del pacchetto (**postinst**) è stata eseguita durante la creazione del filesystem di root. Se lo script restituisce un valore di successo, il pacchetto viene contrassegnato come installato. Se l'esecuzione dello script restituisce un errore, il pacchetto viene contrassegnato come decompresso. Tutti i pacchetti contrassegnati come decompressi hanno i loro script che vengono eseguiti di nuovo immediatamente al primo avvio dell'immagine.

Per aggiungere uno script **postinst**, si può usare il seguente codice come esempio:

```
pkg_postinst_${PN} () {
#!/bin/sh -e
# Insert commands above
}
```

Invece di utilizzare il nome del pacchetto stesso, si può utilizzare la variabile **PN**, che espande automaticamente il nome del pacchetto della ricetta.

Per ritardare l'esecuzione dello script, in modo che venga eseguito sul dispositivo di destinazione stesso, si utilizza la seguente struttura nello script di post-installazione:

```
pkg_postinst_${PN} () {
#!/bin/sh -e
if [ -n "$D" ]; then # AAA: maybe [ -z "$D" ]
# Insert commands here
else
exit 1
fi
}
```

Nell'esempio precedente, si può vedere come viene ritardata l'esecuzione dello script. Se la variabile **\$D** ha un valore, lo script restituisce un errore e il pacchetto viene impostato come decompresso. Significa che qualsiasi comando inserito nella sezione condizionale dell'**if** viene eseguito solo se la variabile **\$D** non è impostata.

Si può anche saltare l'esecuzione postscript, ad esempio al momento della creazione di **rootfs**, per evitare di provare ad avviare un demone in quel momento, ma verificando che sia avviato correttamente quando viene aggiornato nel dispositivo. Ecco un esempio:

```
pkg_postinst_${PN} () {
#!/bin/sh -e
if [ -n "$D" ]; then # AAA: maybe [ -z "$D" ]
exit 0
fi
# Insert commands here to restart
}
```

Quando si genera un'immagine con **read-only-rootfs** in **IMAGE\_FEATURES**, tutti gli script di post-installazione devono essere stati eseguiti con successo. Se un qualsiasi script restituisce un errore e il pacchetto viene impostato solo come decompresso, forzando l'esecuzione dello script dopo la creazione del filesystem root, il task **do\_rootfs** non riesce. Questo controllo durante il build assicura che si identifichi il problema durante la

creazione dell'immagine piuttosto che durante il boot iniziale nel dispositivo target a causa dell'impossibilità di scrivere nel filesystem.

*Assicurarsi che tutta l'esecuzione dello script `pkg_postinst` per i pacchetti installati sia fattibile durante il `do_rootfs`. Questo è richiesto nel caso che `read-only-rootfs` stia in `IMAGE_FEATURES`.*

È importante sottolineare che uno degli errori più comuni durante la creazione di script post-installazione è la mancanza della variabile `D` davanti ai percorsi assoluti. Ciò garantisce che i percorsi siano validi sia nell'ambiente host che in quello target. Inoltre, controllando che la variabile `D` sia vuota consente di determinare quale ambiente viene utilizzato e si può utilizzare codice diverso se in esecuzione durante la generazione del filesystem di root o su un target.

Un altro errore comune consiste nel tentare di eseguire processi specifici o dipendenti dall'architettura target. La soluzione più semplice in questo caso è posticipare l'esecuzione dello script sul target, ma come accennato in precedenza, ciò impedisce l'uso di filesystem di sola lettura.

## La cache di stato condivisa

Il comportamento predefinito di Poky è quello di eseguire il build tutto da zero a meno che BitBake non determini che una ricetta non ha bisogno di essere re-buildata. Il principale vantaggio del build tutto da zero è che il risultato finale è nuovo e non c'è il rischio che i dati precedenti causino problemi. Tuttavia, il build di tutto richiede tempo e risorse di calcolo.

La strategia per determinare se si deve eseguire il build di una ricetta è complessa. Fondamentalmente, BitBake cerca di tenere traccia di quante più informazioni possibili su ogni task, variabile e codice utilizzato nel processo di build. BitBake genera quindi un checksum per tutte le informazioni coinvolte per ogni task.

Poky utilizza tutte queste informazioni fornite da BitBake per archiviare snapshot [istantanee] di tali task come un insieme di dati compressi generati in una cache, chiamata cache di stato condivisa [shared] (`sstate-cache`). Questa cache racchiude il contenuto di ogni output dei task nei pacchetti archiviati nella directory `SSTATE_DIR`. Ogni volta che BitBake si prepara a eseguire un'attività, verificare prima l'esistenza di un pacchetto `sstate-cache` corrispondente. Se il pacchetto è presente, BitBake utilizza il pacchetto precompilato.

Questa è una visione molto semplice dell'intero meccanismo dello "shared state", che è un pezzo di codice piuttosto complesso. Per una panoramica avanzata, si consiglia di leggere la sezione Shared State Cache dello Yocto Project Reference Manual. Per ulteriori informazioni, visitare il seguente link: <http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html>.

Quando si utilizza Poky per diverse build, è necessario tenere a mente che `sstate-cache` deve essere pulita di tanto in tanto poiché continua a crescere man mano che vengono aggiunti sempre più dati memorizzati nella cache per ogni build. C'è un modo semplice per pulirlo, come segue:

```
$ ./scripts/sstate-cache-management.sh --remove-duplicated -d --cache-dir=<path to sstate-cached>
```

Questo rimuove i dati duplicati e quelli vecchi dalla cache.

*Quando si deve rieseguire il build da zero, si deve rimuovere sia `build/tmp` in modo da poter utilizzare `sstate-cache` per velocizzare la build, e rimuovere anche sia `build/tmp` che `sstate-cache` in modo che nessuna cache venga riutilizzata durante la build.*

## Il versioning del package

Il `Package versioning` viene utilizzato per differenziare lo stesso pacchetto in diverse fasi del suo ciclo di vita. Dal punto di vista di Poky, viene utilizzato anche come parte dell'equazione che genera il checksum utilizzato da BitBake per verificare se un task deve essere rieseguito.

La versione del pacchetto, nota anche come `PV`, gioca un ruolo centrale quando si seleziona la ricetta di cui eseguire il build. Il comportamento di default di Poky è quello di preferire sempre la versione più recente della ricetta, a meno che non vi sia una preferenza esplicita diversa, come visto nel [Capitolo 4, Capire il Tool BitBake](#). Ad esempio, si supponga di avere due versioni della ricetta `myrecipe-myrecipe_1.0.bb` e `myrecipe_1.1.bb`. BitBake, per default, esegue il build della ricetta con la versione 1.1.

All'interno della ricetta, si potrebbero avere altre variabili che compongono il versionamento del pacchetto con la variabile `PV`. Questi sono le "package epoch", note come `PE` e i "package revision", noti come `PR`.

La variabile `PE` ha un valore zero di default e viene utilizzata quando si modifica lo schema della versione del pacchetto, interrompendo la possibilità di un normale ordinamento. La "package epoch" è anteposta nella versione del pacchetto, forzando un numero più alto quando necessario. Ad esempio, se un pacchetto utilizza la data per comporre le variabili `PV` come 20140101 e 20140201, lo schema della versione viene modificato per un motivo e viene rilasciata una nuova versione, 1.0, è impossibile determinare se la versione 1.0 è successiva alla versione 20140201. Pertanto, viene utilizzato `PE = "1"`, forzando la versione 1.0 a essere superiore a 20140201 poiché 1:1.0 è maggiore di 0:20140101.

La variabile `PR` ha un valore predefinito di `0` e viene utilizzata come parte del versioning del pacchetto. Quando viene aggiornato, obbliga BitBake a ricostruire tutti i task di una ricetta specifica. Lo si può aggiornare manualmente nei metadati della ricetta per forzare un rebuild che si sa essere necessario. Sebbene l'approccio di impostare manualmente le variabili `PR` all'interno delle ricette possa sembrare interessante, è molto fragile perché si basa sull'interazione e sulla conoscenza umana. A partire da *Yocto Project 1.5 (Dora)*, BitBake utilizza i checksum dei task per controllare ciò che deve essere ricostruito e l'incremento manuale delle `PR` viene utilizzato solo in casi estremamente rari quando il checksum delle attività non cambia.

## La specifica delle dipendenze del pacchetto di runtime

I risultati finali della maggior parte delle ricette sono pacchetti gestiti dal package manager. Come visto nelle sezioni precedenti, richiede informazioni su tutti quei pacchetti e su come si relazionano tra loro. Ad esempio, un pacchetto può dipendere o entrare in conflitto con un altro.

Ci sono più vincoli di pacchetto che devono essere espressi; tuttavia, questi vincoli sono specifici del formato del pacchetto, quindi BitBake ha un set di metadati specifico utilizzato per astrarre quei vincoli.

Di seguito è riportato un elenco dei vincoli di runtime più comunemente utilizzati:

- **RDEPENDS:** Questo è l'elenco dei pacchetti che devono essere installati insieme al pacchetto che lo definisce.
- **RPROVIDES:** Questo è l'elenco dei nomi simbolici forniti da un pacchetto. Per default, un pacchetto include sempre il nome del pacchetto come nome simbolico. Può anche includere altri nomi simbolici forniti in alternativa da quel pacchetto.
- **RCONFLICTS:** Questo è l'elenco dei pacchetti noti per essere in conflitto con questo pacchetto. È possibile installarne solo uno alla volta.
- **RREPLACES:** Questo è un elenco di nomi simbolici che possono essere usati come sostituti di questo pacchetto.

## Feed dei pacchetti

Come visto nel [Capitolo 4, Capire il Tool BitBake](#), i pacchetti svolgono un ruolo centrale, poiché le immagini e gli SDK si basano su di essi. In effetti, `do_rootfs` utilizza un repository di pacchetti locale per recuperare i pacchetti binari durante la generazione del filesystem di root. Questo repository è generalmente noto come *package feed* [feed del pacchetto].

Non vi è alcun motivo per utilizzare questo repository solo per le immagini o per i passaggi di compilazione dell'SDK. In effetti, ci sono diversi validi motivi per rendere questo repository accessibile da remoto internamente nel nostro ambiente di sviluppo o pubblicamente per l'uso sul campo. Alcuni di questi motivi sono i seguenti:

- Testare facilmente un'applicazione aggiornata durante la fase di sviluppo, senza la necessità di una reinstallazione completa del sistema
- Rendere i pacchetti aggiuntivi più flessibili in modo che possano essere installati in un'immagine in esecuzione
- Aggiornare i prodotti sul campo

Per produrre un feed di pacchetto solido, si devono avere incrementi coerenti nella revisione del pacchetto ogni volta che il pacchetto viene modificato. È quasi impossibile farlo manualmente e lo Yocto Project ha un servizio, il *PR service*, appositamente progettato per questo caso.

Il servizio PR, che fa parte di BitBake, viene utilizzato per incrementare il `PR` senza l'interazione umana ogni volta che BitBake rileva una modifica del checksum in un task. In sostanza, inserisce un suffisso in `PR` nel formato `${PR}.x`. Ad esempio, considerando `PR = "r34"` dopo le successive interazioni del servizio PR, il valore `PR` diventa `r34.1`, `r34.2`, `r34.3` e così via.

Quando si utilizza Poky per generare immagini e non si ha come target un feed di pacchetto remoto, il servizio PR non è richiesto perché BitBake attiva i rebuild richiesti a causa delle modifiche al checksum del task. L'uso del PR service è fondamentale per dei solidi feed di pacchetti, poiché richiede l'aumento della versione in modo lineare.

>Anche se si dovrebbe utilizzare il servizio PR per avere un controllo delle versioni, ciò non preclude la necessità di impostare PR manualmente in casi eccezionali.

Per default, il PR service non è abilitato né in esecuzione. Per consentirne l'esecuzione in locale, si deve impostare la variabile `PRSERV_HOST` nella configurazione di BitBake, ad esempio, in `build/conf/local.conf`, come segue:

```
PRSERV_HOST = "localhost:0"
```

Questo approccio è adeguato quando il build avviene su un singolo computer, che compila ogni pacchetto del feed di pacchetto. BitBake avvia e arresta il server a ogni build e aumenta automaticamente i valori `PR` richiesti.

Per una configurazione più complessa, con più computer che lavorano su un feed di pacchetto comune e condiviso, è necessario disporre di un unico servizio PR in esecuzione, che viene utilizzato da tutti i sistemi di costruzione associati al feed. In questo caso, si deve avviare il servizio PR nel server utilizzando il comando `bitbake-prserv`, come mostrato qui:

```
$ bitbake-prserv --host <ip> --port <port> --start
```

Oltre ad avviare manualmente il servizio, è necessario aggiornare il file di configurazione di BitBake (ad esempio `build/conf/local.conf`) di ciascun sistema di build, che si connette a un server utilizzando la variabile `PRSERV_HOST` come descritto in precedenza in modo che ogni sistema punti all'IP e alla porta del server.

## Utilizzo dei feed dei pacchetti

Per utilizzare i feed dei pacchetti, i seguenti due componenti devono essere configurati correttamente:

- Il server, per fornire l'accesso ai pacchetti
- Il client, per accedere al server e scaricare i pacchetti richiesti

L'insieme dei pacchetti offerti dal feed dei pacchetti è determinato dalle ricette da buildare. Si può eseguire il build di una o più ricette e proporle, oppure si può eseguire il build di un insieme di immagini per generare il pacchetto desiderato. Una volta che soddisfatti dell'insieme dei pacchetti offerti, si deve creare l'indice dei pacchetti che devono essere forniti dai feed. Questo viene eseguito dal seguente comando:

```
$ bitbake package-index
```

I pacchetti sono disponibili all'interno della directory `build/tmp/deploy`. Si deve scegliere la rispettiva sottodirectory a seconda del formato del pacchetto. Poky utilizza RPM per default, quindi si deve esportare il contenuto della directory `build/tmp/deploy/rpm`.

*Si deve eseguire `bitbake package-index` dopo il building di tutti i pacchetti altrimenti non verranno inclusi nel "package database".*

L'indice del pacchetto, insieme ai pacchetti, deve essere reso disponibile tramite un protocollo di trasferimento come HTTP. Si può utilizzare qualsiasi server per questo compito, come Apache, Nginx, Lighttpd e altri. Un modo conveniente per rendere disponibili i pacchetti tramite HTTP per lo sviluppo locale è utilizzare il semplice server Python, come mostrato qui:

```
$ cd build/tmp/deploy/rpm
$ python3 -m http.server 5678
```

Per aggiungere il supporto per la gestione dei pacchetti all'immagine, ci sono un paio di modifiche da fare. Si deve aggiungere `package-management` in `EXTRA_IMAGE_FEATURES` e impostare l'URI per il recupero dei pacchetti su `PACKAGE_FEED_URI`. Ad esempio, si può aggiungere questo a `build/conf/local.conf`:

```
PACKAGE_FEED_URI = "http://my-ip-address:5678"
EXTRA_IMAGE_FEATURES += " package-management "
```

Le variabili `IMAGE_FEATURES` e `EXTRA_IMAGE_FEATURES` variables vengono descritte in dettaglio nel [Capitolo 11, Creazione di Layer Personalizzati](#). Se si vuole un'immagine piccola senza supporto per la gestione dei pacchetti, non si deve includere `package-management` in `EXTRA_IMAGE_FEATURES`.

Le configurazioni `PACKAGE_FEED_URI` e `EXTRA_IMAGE_FEATURES` garantiscono che l'immagine in esecuzione sul lato client possa accedere al server e disponga dei tool necessari per installare, rimuovere e aggiornare i suoi pacchetti.

Dopo aver eseguito questi passaggi, so è in grado di utilizzare la gestione dei pacchetti di runtime nel dispositivo target.

Ad esempio, se si sceglie il formato RPM per l'immagine, si possono recuperare le informazioni sul repository utilizzando il seguente comando:

```
$ dnf check-update
```

Si usano i comandi `dnf search <package>` e `dnf install <package>` per trovare e installare i pacchetti dai repository.

A seconda del formato scelto, i comandi per il target per aggiornare l'indice del pacchetto, cercare e installare un pacchetto sono diversi. Vedere le righe di comando disponibili per ogni formato nella tabella seguente:

Formato del package	RPM	IPK	DEB
Aggiornamento dell'indice dei package	<code>dnf check-updates</code>	<code>opkg update</code>	<code>apt-get update</code>
Ricerca di un package	<code>dnf search &lt;package&gt;</code>	<code>opkg search &lt;package&gt;</code>	<code>apt-cache search &lt;package&gt;</code>
Installazione di un package	<code>dnf install &lt;package&gt;</code>	<code>opkg install &lt;package&gt;</code>	<code>apt-get install &lt;package&gt;</code>
Aggiornamento del sistema	<code>dnf upgrade</code>	<code>opkg upgrade</code>	<code>apt-get dist-upgrade</code>

I passaggi mostrati in questo capitolo sono ottimali per l'uso in una fase di sviluppo locale perché consentono di installare i pacchetti in un'immagine già distribuita.

Il feed dei pacchetti per l'aggiornamento del sistema sul campo richiede un'enorme quantità di lavoro per testare tutti i diversi scenari di aggiornamento e garantire che il sistema non vada in uno stato anomalo. Di solito, gli aggiornamenti dell'immagine completa vengono preferiti per l'uso in produzione.

La gestione di un feed è molto più complessa e coinvolge molti altri aspetti come le catene di dipendenza dei pacchetti, diversi scenari di aggiornamento e altro ancora. La creazione di un server esterno di feed di pacchetti complessi non rientra negli scopi di questo libro, quindi fare riferimento alla documentazione dello Yocto Project per ulteriori dettagli.

## Sommario

Questo capitolo ha illustrato il concetto di packaging, che ha un ruolo centrale per Poky e BitBake; versionamento dei pacchetti; e come questo influisce sul comportamento di Poky durante il re-building e il feed di pacchetti. Ha anche mostrato come configurare un'immagine da aggiornare usando i pacchetti precompilati forniti da un server remoto.

Nel prossimo capitolo si vedrà la sintassi dei metadati BitBake e i suoi operatori per aggiungere (**append**), anteporre (**prepend**) e rimuovere (**remove**) del contenuto dalle variabili, le espansioni delle variabili e così via. Si sarà quindi in grado di comprendere meglio il linguaggio utilizzato nei motori dello Yocto Project.

## 7. Approfondimenti sui Metadati di BitBake

A questo punto, sappiamo come generare immagini e pacchetti, nonché come utilizzare i feed dei pacchetti, praticamente tutto ciò che dobbiamo sapere per un semplice utilizzo di Poky. Di seguito, vedremo come controllare il comportamento di Poky per raggiungere i nostri obiettivi e ottenere il massimo beneficio dallo Yocto Project nel suo insieme.

In questo capitolo miglioreremo la nostra comprensione della sintassi dei metadati di BitBake. Impareremo a usare gli operatori `append`, `prepend` e `remove` per modificare il contenuto di variabili, espansioni di variabili e così via. Questi sono i concetti chiave utilizzabili per creare le ricette e le personalizzazioni che vedremo nel [Capitolo 10](#), *Esplorazione di Layer Esterni*, nel [Capitolo 11](#), *Creazione di Layer Personalizzati* e nel [Capitolo 12](#), *Personalizzazione delle ricette esistenti*.

### Utilizzo dei metadati

La quantità di metadati utilizzati da BitBake è enorme. Per ottenere il massimo vantaggio dall'utilizzo di Poky, si devono dominare. Come visto nel [Capitolo 4](#), *Capire il Tool BitBake*, i metadati si possono classificare nelle seguenti tre aree:

- **Configurazione** (i file `.conf`): I file di configurazione definiscono il contenuto globale utilizzato per fornire informazioni e configurare il funzionamento delle classi e delle ricette
- **Classi** (i file `.bbclass`): Le classi sono disponibili per l'intero sistema e possono essere ereditate dalle ricette per facilitare la manutenzione ed evitare la duplicazione del codice
- **Ricette** (i file `.bb` e `.bbappend`): Le ricette descrivono i task [attività] da eseguire e forniscono le informazioni richieste per consentire a BitBake di generare la catena di task richiesta. È il tipo di metadati più comunemente usato in quanto sono i punti in cui si mette tutto al lavoro. I tipi più comuni di ricette generano pacchetti e immagini.

Le classi e le ricette sono scritte in un mix di Python e codice di shell scripting. Quando una ricetta viene eseguita da BitBake, viene creato uno stato locale in modo che le classi ereditate e i metadati specifici della ricetta non producano effetti collaterali altrove.

### Lavorare con i metadati

La sintassi utilizzata dai metadati BitBake può essere fuorviante e talvolta può essere difficile da tracciare. Possiamo controllare il valore di ogni variabile che vogliamo usando l'opzione dell'ambiente (`-e` or `--environment`) di BitBake, ad esempio:

```
$ bitbake -e <recipe> | grep <variable>
```

Per comprendere più in dettaglio come funziona BitBake, fare riferimento a <https://www.yoctoproject.org/docs/current/bitbake-user-manual/bitbake-user-manual.html>.

Nelle sezioni seguenti vedremo la maggior parte della sintassi comunemente usata nelle ricette.

### L'impostazione della variabile di base

L'assegnazione di una variabile può essere eseguita come illustrato:

```
FOO = "bar"
```

Nell'esempio precedente, il valore della variabile `foo` è `bar`.

L'assegnazione delle variabili è fondamentale per la sintassi dei metadati BitBake poiché la maggior parte degli altri esempi viene eseguita utilizzando variabili.



## Espansione della variabile

BitBake supporta il riferimento alle variabili. La sintassi è molto simile a di Shell Script, ad esempio:

```
A = "aval"
B = "pre${A}post"
```

Ciò si traduce in **A** contenente `aval` e **B** contenente `preavalpost`. Una cosa importante da tenere a mente è che la variabile si espande solo quando viene effettivamente utilizzata, come mostrato:

```
A = "aval"
B = "pre${A}post"
A = "change"
```

L'esempio precedente illustra la *pigrizia* della valutazione di BitBake. Quando viene utilizzato **B**, valuta **A** (poiché ne è un riferimento); **A** ora contiene una modifica e **B** è `prechange`post.

## Impostazione di un valore di default utilizzando ?=

Quando è necessario fornire un valore di default, è possibile utilizzare l'operatore `?=`. Il codice seguente ne mostra l'utilizzo:

```
A ?= "aval"
```

Se **A** viene impostato prima che venga chiamato il codice precedente, mantiene il suo valore precedente; se **A** non è stato precedentemente impostato, viene impostato su `aval`. In sostanza, l'operatore `?=` assegna un nuovo valore ad una variabile se non ne è già stata impostata una.

Si noti che se sono presenti più `?=` assegnazioni a una singola variabile, viene utilizzata la prima di queste. Ad esempio, potremmo avere quanto segue:

```
A ?= "aval"
A ?= "change"
```

La variabile **A** ha il valore `aval`. Tuttavia, se abbiamo precedentemente impostato **A**, viene utilizzato. Per esempio:

```
A = "before"
A ?= "aval"
A ?= "change"
```

La variabile **A** viene mantenuta come `before`.

## Impostazione di un valore di default utilizzando ??=

Un altro modo per fornire un valore di default consiste nell'usare l'operatore `??=`. La differenza tra `??=` e `?=` è che con `??=` l'assegnazione non si verifica fino alla fine del processo di analisi in modo che venga utilizzata l'ultima assegnazione anziché la prima `??=` a una determinata variabile. Controllare il seguente codice:

```
A ??= "somevalue"
A ??= "someothervalue"
```

Se **A** è impostato prima del codice precedente, mantiene il valore. Se **A** non è stato impostato in precedenza, viene impostato su `someothervalue`.

## Espansione immediata della variabile

Utilizzare l'operatore `:=` quando è necessario forzare l'espansione immediata di una variabile. Ne risulta che il contenuto della variabile viene espanso immediatamente anziché quando la variabile viene effettivamente utilizzata, come segue:

```
T = "123"
A := "${B} ${A} test ${T}"
B = "${T} bval"
T = "456"
C = "cval"
C := "${C}append"
```

Alla fine di questo esempio, **A** conterrà il test `test 123`, **B** conterrà `456 bval` e **C** sarà `cvalappend`. Quando **A** viene espanso, **B** non è ancora definito, quindi **B** è vuoto.

## Accodare e anteporre

BitBake offre due set di operatori per l'accodamento e il pre-inserimento. I primi sono += e =+. Il codice seguente ne illustra l'utilizzo:

```
B = "bval"
B += "additionaldata"
C = "cval"
C =+ "test"
```

Alla fine di questo esempio, **B** contiene **bval additionaldata** e **C** contiene **test cval**. È importante notare che questi operatori includono uno spazio aggiuntivo tra ogni chiamata.

Quando vogliamo evitare di aggiungere spazi, utilizziamo gli operatori .= e =.. Guardiamo il seguente codice:

```
B = "bval"
B .= "additionaldata"
C = "cval"
C =. "test"
```

In questo esempio, **B** è ora **bvaladditionaldata** e **C** è **testcval**. Contrariamente all'esempio precedente, gli operatori .= e =. non aggiungono uno spazio aggiuntivo. In genere, gli operatori += e =+ vengono utilizzati per aggiungere elementi agli elenchi, mentre gli operatori .= e =. vengono utilizzati per concatenare le stringhe.

## Sintassi degli operatori di override

Si può anche aggiungere e anteporre il valore di una variabile usando una sintassi in stile override. Per esempio:

```
B = "bval"
B_append = "additionaldata"
C = "cval"
C_prepend = "test"
```

In questo esempio, **B** è ora **bvaladditionaldata** e **C** è **testcval**. Questi operatori non aggiungono uno spazio aggiuntivo.

C'è una sottile differenza tra il modo in cui vengono analizzati gli operatori di accodamento e pre-inserimento. Quando si utilizzano gli operatori estesi, l'espansione delle variabili viene forzata prima dell'esecuzione dell'operazione, ad esempio:

```
A ?= "aval"
A .= "more"
B ?= "bval"
B_prepend = "more"
```

Nell'esempio precedente, quando viene eseguito l'operatore .=, **A** non viene espanso e quindi **A** diventa **more**. Nel caso di **B**, prima di eseguire l'operazione di **prepend**, **B** viene espanso in modo che diventi **morebval**.

Oltre ad accodare e anteporre i contenuti, possiamo anche voler rimuovere il valore di una variabile, cosa che può essere fatta come segue:

```
A = "aval1 aval2 aval3"
A_remove = "aval1 aval3"
```

In questo esempio, il valore **A** è ora **aval2**. L'operatore **remove** considera il valore della variabile come un elenco di stringhe separate da spazi, quindi l'operatore può rimuovere una o più stringhe da questo elenco.

## Set di metadati condizionali

BitBake fornisce un modo molto facile da usare per scrivere metadati condizionali. È fatto con un meccanismo chiamato *overrides*.

La variabile **OVERRIDES** contiene valori separati da due punti (:) e ogni valore è un elemento che deve soddisfare delle condizioni. Quindi, se abbiamo una variabile condizionale su **arm** e **arm** è in **OVERRIDES**, viene utilizzata la versione della variabile specifica per **arm** anziché la versione non condizionale, come mostrato:

```

OVERRIDES = "architecture:os:machine"
TEST = "defaultvalue"
TEST_os = "osspecificvalue"
TEST_other = "othercondvalue"

```

In questo esempio, **TEST** sarà **osspecificvalue** a causa della condizione di **os** in **OVERRIDES**.

## Accodamento condizionale

BitBake supporta anche l'accodamento e l'anteposizione alle variabili in base al fatto che qualcosa sia in **OVERRIDES**, come mostrato qui:

```

DEPENDS = "glibc ncurses"
OVERRIDES = "machine:local"
DEPENDS_append_machine = " libmad"

```

Nell'esempio precedente, **DEPENDS** è impostato su **glibc ncurses libmad**.

## Inclusione di file

BitBake fornisce due direttive per l'inclusione dei file: **include** e **require**.

Con la prima direttiva, **include**, BitBake tenta di inserire il file in quella posizione. Se il path specificato sulla riga dell'**include** è relativo, BitBake individua la prima istanza che trova all'interno di **BBPATH**. Se non è possibile trovare il file di riferimento, l'analisi non fallisce.

Al contrario, la seconda direttiva, **require**, solleva un errore **ParseError** se non è possibile trovare il file da includere. In tutti gli altri aspetti, si comporta proprio come la direttiva **include**.

La convenzione normalmente adottata nello Yocto Project è quella di utilizzare un file **.inc** per condividere il codice comune tra due o più ricette.

## Espansione delle variabili Python

BitBake semplifica l'espansione delle variabili Python con la seguente sintassi:

```
VARIABLE = "${@<python-command>}"
```

Ciò offre un'enorme flessibilità all'utente, come si può vedere nel seguente esempio:

```
DATE = "${@time.strftime('%Y%m%d',time.gmtime())}"
```

Ciò si traduce nella variabile **DATE** contenente la data odierna.

## Definizione dei metadati eseguibili

Le ricette dei metadati (**.bb**) e dei file delle classi (**.bbclass**) possono usare codice Shell Script nel seguente modo:

```

do_mytask () {
    echo "Hello, world!"
}

```

Questo è essenzialmente identico all'impostazione di una variabile, tranne per il fatto che questa variabile sembra essere un codice shell eseguibile. È importante fare attenzione quando si scrive il codice script shell poiché non dovremmo usare codice specifico come **bash** o **zsh**. In caso di dubbio, un buon modo per verificare se il nostro codice è sicuro è utilizzare la **dash** shell per testarlo.

Un altro modo per iniettare il codice è usare il codice Python, come mostrato:

```

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

```

Questo è simile al codice precedente ma lo contrassegna come Python in modo che BitBake sappia che è codice Python e lo esegue di conseguenza.

## Definizione delle funzioni Python nello spazio dei nomi globale

Potrebbe essere necessario utilizzare le funzioni Python per generare un valore per una variabile o per qualche altro uso. Questo può essere fatto facilmente, nelle ricette (`.bb`) e nelle classi (`.bbclass`), usando il seguente codice:

```
def get_depends(d):
    if d.getVar('SOMECONDITION'):
        return "dependencywithcond"
    else:
        return "dependency"
SOMECONDITION = "1"
DEPENDS = "${@get_depends(d)}"
```

Di solito, si deve accedere al database BitBake quando si scrive una funzione Python. Una convenzione tra tutti i metadati è che la variabile `d` viene utilizzata per puntare al database di BitBake e di solito viene passata come ultimo parametro di una funzione.

Quindi, nell'esempio precedente, chiediamo al database il valore della variabile `SOMECONDITION` e restituiamo un valore che dipende da esso.

Il risultato dell'esempio è che `DEPENDS` contiene `dependencywithcond`.

## Il sistema dell'ereditarietà

La direttiva `inherit` fornisce i mezzi per specificare quali classi di funzionalità richiede la ricetta (`.bb`). È una forma rudimentale di ereditarietà. Ad esempio, possiamo facilmente astrarre i task coinvolti nell'uso di `autoconf` e `automake`, e inserirli in `.bbclass` affinché le ricette possano farne uso. Una determinata `.bbclass` si trova cercando `classes/filename.bbclass` in `BBPATH`, dove `filename` è ciò che abbiamo ereditato. Quindi, in una ricetta che utilizza `autoconf` o `automake`, possiamo usare quanto segue:

```
inherit autotools
```

Questo indica a BitBake di ereditare `autotools.bbclass`, quindi fornisce i task di default che funzionano bene per la maggior parte dei progetti basati su `autoconf` o `automake`.

## Sommario

In questo capitolo abbiamo appreso in dettaglio la sintassi dei metadati BitBake; i suoi operatori per `accodare` (`append`), `anteporre` (`prepend`) e `remove` (`rimuovere`), contenuto dalle variabili; espansioni di variabili; e così via, inclusi alcuni esempi di utilizzo.

Nel prossimo capitolo impareremo come usare Poky per creare strumenti di compilazione esterni e produrre un filesystem di root adatto allo sviluppo in un target. Verrà inoltre spiegato il possibile utilizzo dell'integrazione con *Eclipse*.

## 8. Sviluppo con lo Yocto Project

Finora abbiamo usato Poky come strumento di build; in altre parole, lo abbiamo utilizzato come strumento per progettare e generare l'immagine che verrà utilizzata sui prodotti. In questo capitolo vedremo come il tool può aiutarci con lo sviluppo di applicazioni o del kernel, creare strumenti di compilazione esterni, produrre un filesystem root adatto per "cross development" e generare un'immagine con gli strumenti da usare nella macchina target per lo sviluppo.

### ***Decifrare il "software development kit"***

Un `software development kit` (SDK) è un insieme di strumenti e file utilizzati per lo sviluppo e il debug. Questi strumenti includono compilatori, linker, debugger, header di librerie esterne e file binari e possono includere utilità e applicazioni personalizzate. Questo insieme di strumenti di programmazione è chiamato toolchain.

Nello sviluppo embedded, la toolchain è spesso composta da strumenti multi-piattaforma o strumenti eseguiti su un'architettura che poi produce un binario da utilizzare in un'altra architettura. Ad esempio, un binario `gcc` che viene eseguito su una macchina compatibile con x86-64 e produce un binario per una macchina ARM è un cross-compilatore. Quando il tool e il file binario risultante vengono eseguiti sulla stessa architettura, si parla di build nativa.

Di solito, quando si lavora su codice sorgente e si personalizzano e utilizzano librerie esterne, ad esempio `libusb` o `libgl`, tali librerie vengono utilizzate per il build in fase di runtime [esecuzione]. Il sorgente personalizzato può essere compilato rispetto ai file header della libreria e il file binario può essere spostato da qualche parte durante l'esecuzione. L'insieme di file utilizzati in fase di build è collocato nella directory `sysroots`, parte dell'SDK di Poky, che è molto configurabile, a seconda dell'applicazione, ed è molto semplice per un uso generale.

Quando Poky esegue task tramite l'uso di BitBake, ha anche bisogno di una toolchain per poter compilare e linkare i file binari per il target. Questa è detta toolchain interna perché i tool vengono utilizzati internamente dal sistema di build. Questi strumenti non sono destinati all'uso esterno in quanto non sono preparati a tale scopo, sebbene possano essere utilizzati in alcuni casi d'uso molto specifici e avanzati.

### ***Lavorare con l'SDK di Poky***

Di solito, il flusso di lavoro standard di Poky include la creazione di ricette e immagini di pacchetti e la decisione di cosa verrà installato sull'immagine del prodotto finale. Tuttavia, viene impiegata un'enorme quantità di tempo per sviluppare, scrivere, testare o adattare il codice sorgente per la nostra applicazione.

Quando scriviamo e testiamo la nostra applicazione, ci preoccupiamo solo dell'applicazione stessa, fornendo le librerie che l'applicazione richiede prima dello sviluppo dell'applicazione, sebbene questo possa essere un processo iterativo. Tuttavia, vogliamo un ambiente di test che sia il più simile possibile a quello finale, principalmente per la compatibilità della toolchain ma anche per evitare cambiamenti comportamentali quando integriamo l'applicazione nel nostro prodotto.

Per questo compito, possiamo creare una toolchain da utilizzare esternamente con l'ambiente Poky. Poky genera un pacchetto SDK installabile su qualsiasi computer, indipendentemente dal fatto che Poky sia installato su di esso. Inoltre, la toolchain installata è compatibile con quella interna. Oltre alla toolchain, l'SDK può anche fornire una serie di file di libreria (come binari di libreria e file header), a seconda delle esigenze.

## Utilizzo di un SDK basato su immagini

Per il codice sorgente personalizzato, conosciamo le librerie di dipendenze e le altre applicazioni da cui dipendiamo. In questi casi, si può creare un'immagine che rifletta esattamente le nostre esigenze o utilizzare l'immagine più vicina fornita da Poky.

Per creare l'SDK basato su immagini, eseguire il comando seguente:

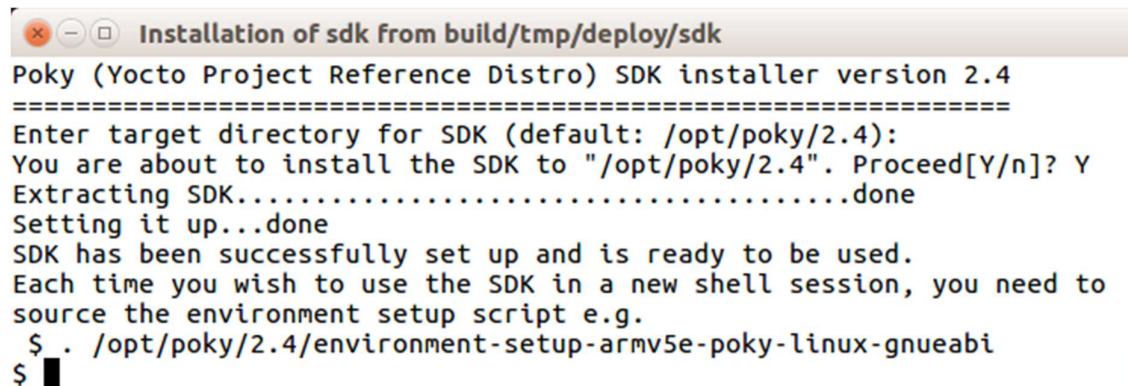
```
$ bitbake core-image-full-cmdline -c populate_sdk
```

Con questo comando, l'SDK viene creato in base all'immagine `core-image-full-cmdline`. Se abbiamo un'immagine personalizzata, possiamo usarla al suo posto. L'SDK viene generato per corrispondere all'architettura della macchina che è stata impostata utilizzando la variabile `MACHINE`.

Dopo il build dell'SDK, è possibile trovare uno script binario in `build/tmp/deploy/sdk/poky-glibc-x86_64-core-image-full-cmdline-armv5e-toolchain-2.4.sh`.

Non aprire lo script precedente su un semplice editor di testo. Lo script contiene un pezzo di codice binario che potrebbe causare far crashare un editor di testo.

Lo script risultante deve essere installato prima di essere utilizzato. Possiamo vedere il processo di installazione nel seguente screenshot:



```
Installation of sdk from build/tmp/deploy/sdk
Poky (Yocto Project Reference Distro) SDK installer version 2.4
=====
Enter target directory for SDK (default: /opt/poky/2.4):
You are about to install the SDK to "/opt/poky/2.4". Proceed[Y/n]? Y
Extracting SDK.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to
source the environment setup script e.g.
$ . /opt/poky/2.4/environment-setup-armv5e-poky-linux-gnueabi
$
```

Nell'esempio precedente, la directory di installazione è `/opt/poky/2.4`; si può, tuttavia, scegliere qualsiasi directory. L'installazione fornisce quanto segue:

- `environment-setup-armv5te-poky-linux-gnueabi`: Questo è lo script utilizzato per impostare tutte le variabili di ambiente necessarie per utilizzare la toolchain.
- `site-config-armv5te-poky-linux-gnueabi`: Questo è il file con le variabili utilizzate durante la creazione della toolchain.
- `version-armv5te-poky-linux-gnueabi`: Queste sono le informazioni sulla versione e sul timestamp.
- `sysroots`: Questa è una copia della directory `rootfs` delle immagini utilizzate per la generazione dell'SDK. Include file binari, header e i file di libreria distribuiti in sottodirectory come:
  - `armv5te-poky-linux-gnueabi`: Contiene file per macchine ARM
  - `x86_64-pokysdk-linux`: Questi sono file per macchine con compatibilità x86-64

## SDK generico – meta-toolchain

Un'altra opzione è creare un SDK generico, ma con cross-compilatore, strumenti di debug e un set di base di librerie e file di header. Questo SDK generico è chiamato `meta-toolchain` e viene utilizzato principalmente per lo sviluppo di kernel e bootloader e per il processo di debug. Per crearlo, utilizzare il seguente comando:

```
$ bitbake meta-toolchain
```

Il file risultante è `build/tmp/deploy/sdk/poky-eglibc-x86_64-meta-toolchain-armv5te-toolchain-2.4.sh` per la macchina `qemuarm`. Il processo di installazione è esattamente lo stesso dell'SDK basato su immagini.

Sebbene questo SDK sia molto utile, si consiglia vivamente di creare un'immagine personalizzata che soddisfi le esigenze della nostra applicazione, poi creare l'SDK basato su questo.

## Utilizzo di un SDK

Per utilizzare un SDK per creare un'applicazione personalizzata, ad esempio `hello-world.c`, possiamo utilizzare le seguenti righe, destinate all'architettura ARM:

```
$ source /opt/poky/2.4/environment-setup-armv5e-poky-linux-gnueabi
$ ${CC} hello-world.c -o hello-world
```

Per confermare che il file binario generato è stato creato correttamente per l'architettura di destinazione, possiamo utilizzare l'utilità del file come segue:

```
$ file hello-world
hello-world: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux.so.3, for GNU/Linux 3.2.0, BuildID[sha1]=ce9b7de598f76a9611f98a4ce6b1af6018c0471b, not
stripped
```

Un altro progetto molto comunemente usato è il kernel Linux. Il kernel Linux utilizza l'utilità `LD` per il link, quindi è necessario reimpostare la variabile `LDFLAGS` utilizzando il comando `unset` in modo che torni al suo valore predefinito, come definito per l'uso con GCC.

Quando si vuole il build del codice sorgente del kernel Linux, possiamo usare la seguente sequenza di comandi:

```
$ source /opt/poky/2.4/environment-setup-armv5e-poky-linux-gnueabi
$ unset LDFLAGS
$ make defconfig
$ make zImage
```

L'Extensible SDK (eSDK) dello Yocto Project consente lo sviluppo distribuito in quanto gli sviluppatori possono aggiornare ed estendere l'ambiente SDK esistente durante la vita del progetto. Sono necessarie alcune impostazioni dell'infrastruttura per l'uso corretto di eSDK come mirror di `state-cache` e server eSDK, che richiedono una configurazione complessa che va oltre lo scopo di questo libro. Fare riferimento a <http://www.yoctoproject.org/docs/2.4/sdk-manual/sdk-manual.html>

## Sviluppo di applicazioni sul target

Per i sistemi embedded, esiste una parte di debug che dovrebbe essere eseguita su un hardware reale perché l'applicazione utilizza periferiche specifiche dell'hardware e potrebbe essere difficile emularle. Inoltre, alcuni dei processi di debug dipendono dai segnali sorgente, elettrici o ottici, altrimenti l'azione risultante sarà un comportamento meccanico che potrebbe essere difficile da testare efficacemente sugli emulatori.

Il primo scenario proposto per lo sviluppo sul sistema target è quello di creare un'immagine di sviluppo da utilizzare insieme a una toolchain esterna. Un'immagine di sviluppo viene riempita con file header e link a librerie aggiuntive. Questa sarà un'immagine preparata per fornire un ambiente di compilazione per un'applicazione personalizzata, utilizzabile con una toolchain personalizzata o la toolchain esterna dello Yocto Project. La riga seguente aggiunge queste proprietà a un'immagine:

```
IMAGE_FEATURES += "dev-pkgs"
```

Nel caso in cui si desideri modificare solo `build/conf/local.conf`, la variabile da utilizzare è `EXTRA_IMAGE_FEATURES`. La variabile `IMAGE_FEATURES` è descritta meglio nel [Capitolo 11, Creazione di Layer Personalizzati](#).

L'immagine risultante include file header e link a librerie aggiuntive e può essere utilizzata durante il ciclo di sviluppo dell'applicazione personalizzata. Il build dell'applicazione personalizzata può essere fatto su questa immagine del file system root, il che significa che l'immagine stessa non deve essere ricreata ogni volta. Inoltre, l'immagine può essere

condivisa tra tutti gli sviluppatori che lavorano allo stesso progetto. Ognuno avrà una copia e tutti staranno sulla stessa pagina.

*Le funzionalità di dev-pkgs installano tutti i pacchetti \${PN}-dev nell'immagine.*

Per gli sviluppatori che preferiscono o necessitano di utilizzare una build nativa invece di creare una build di sviluppo, Poky può essere configurato per generare un'immagine SDK. Tale immagine contiene la toolchain e i pacchetti di sviluppo (file header e link alle librerie). Significa che possiamo creare, eseguire, testare ed eseguire il debug del codice sorgente della nostra applicazione personalizzata sul computer target.

Per aggiungere gli strumenti di sviluppo all'interno di un'immagine, si deve includere la funzionalità `tools-sdk` nella variabile `IMAGE_FEATURES`. Dovremmo usare `EXTRA_IMAGE_FEATURES` se viene aggiunto in `build/conf/local.conf`.

*Da ricordare che ultimamente ci sono nuovi processori sviluppati per mercati specifici che possono fornire set di risorse completamente differenti (elaborazione, periferiche, memoria e così via).*

Una build nativa è una buona opzione quando abbiamo un microprocessore che fornisce prestazioni ragionevoli e il nostro dispositivo ha memoria sufficiente per rendere fattibile la build nativa. Le risorse necessarie per una build nativa variano notevolmente da una libreria o un'applicazione all'altra.

## Integrazione con Eclipse

Eclipse è un IDE molto potente ed è ampiamente utilizzato per lo sviluppo e il debug di applicazioni personalizzate. Può essere configurato per funzionare con l'SDK di Poky. Nella "Yocto Project SDK Developer's Guide" su <http://www.yoctoproject.org/docs/2.4/sdk-manual/sdk-manual.html>, possiamo trovare la versione Eclipse supportata e possiamo imparare come per configurarlo. Nel manuale sono inclusi Yocto Project ADT e un'immagine basata sull'integrazione di toolchain generica.

Non appena il nostro Eclipse è configurato, possiamo usarlo per lo sviluppo. Possiamo utilizzare l'IDE per scrivere il codice sorgente e la toolchain Poky può essere utilizzata per compilarlo in modo incrociato, poiché Eclipse supporta l'uso di questa toolchain esterna.

Inoltre, possiamo utilizzare Eclipse per distribuire il file binario generato sul target, connesso con Eclipse tramite Ethernet. Il file binario e qualsiasi altro artefatto richiesto vengono copiati nel filesystem root del target ed è possibile utilizzare il filesystem subito dopo il trasferimento.

Non appena il binario viene copiato nel filesystem di root, possiamo utilizzare Eclipse per eseguire il debug dell'applicazione. Ciò significa che siamo autorizzati a utilizzare il debug passo-passo, con il binario eseguito direttamente sulla macchina di destinazione.

Per realizzare l'integrazione di Eclipse, è importante includere la funzione Eclipse nell'immagine di destinazione aggiungendo la seguente parte di codice in `build/conf/local.conf`:

```
IMAGE_FEATURES += "eclipse-debug"
```

Gli autori di questo libro hanno deciso di non includere una guida passo passo su come installare e configurare Eclipse perché richiede diversi passaggi e diventerà obsoleto molto velocemente. Il sito web <http://www.yoctoproject.org/docs/2.4/sdk-manual/sdk-manual.html>, invece, contiene una completa e tutorial aggiornato per questo.



## Sommario

In questo capitolo abbiamo appreso che lo Yocto Project può essere utilizzato sia per lo sviluppo che per la creazione di immagini. Abbiamo imparato come creare due tipi di toolchain, basate su immagini e generiche, come usarle e come creare un'immagine di sviluppo per creare e distribuire la nostra applicazione sulla macchina target. Inoltre, abbiamo imparato come utilizzare Eclipse nella fase di sviluppo per scrivere, creare ed eseguire il debug delle nostre applicazioni.

Nel prossimo capitolo, vedremo come si può configurare Poky per il processo di debug, come configurare il sistema per fornire gli strumenti necessari per un debug remoto usando GDB, come tenere traccia delle modifiche usando `buildhistory` e come usare un pratico strumento chiamato `devshell`.

## 9. Debugging con lo Yocto Project

Il processo di debug è un passaggio importante in ogni ciclo di sviluppo. In questo capitolo capiremo come configurare Poky per il processo di debug, ad esempio, come possiamo configurare il nostro sistema per fornire gli strumenti necessari per un debug remoto usando GDB, come possiamo tenere traccia delle nostre modifiche usando `buildhistory` e come possiamo usare un pratico strumento chiamato devshell.

### ***Differenziazione dei metadati e del debug delle applicazioni***

Quando pensiamo per la prima volta al debug, di solito non ci rendiamo conto che esistono diversi tipi di debug.

Il debug dei metadati è necessario per garantire che il comportamento dei task di BitBake sia in linea con gli obiettivi e per identificare il colpevole quando non lo è. In questo caso, utilizziamo diversi file di log generati da BitBake nell'host per aiutare a tracciare il path di esecuzione del task coinvolto. In conseguenza di un comportamento errato, un file potrebbe non essere copiato o una funzionalità potrebbe non essere abilitata.

D'altra parte, il debug del codice di runtime è più naturale per noi in quanto è essenzialmente lo stesso che facciamo durante il normale ciclo di sviluppo di un'applicazione, una libreria o un kernel. A seconda del tipo di problema che stiamo cercando, lo strumento giusto può variare da un debugger a codice di strumentazione (ad esempio, l'aggiunta di stampe di debug).

In questo capitolo, descriviamo in dettaglio il debug dei metadati in quanto è l'essenza dello Yocto Project e ci supporta durante lo sviluppo e l'uso di Poky.

### ***Tracking dell'immagine, pacchetti e contenuto dell'SDK***

Il modo più semplice per avere l'immagine, i pacchetti e l'SDK, insieme ai contenuti previsti, è utilizzare il meccanismo di `Build History`.

Quando una ricetta viene aggiornata per una nuova versione o cambia il suo codice, può influenzare i contenuti inseriti nei pacchetti generati e, di conseguenza, nell'immagine o nell'SDK.

Poiché Poky gestisce un'enorme quantità di ricette e le nostre immagini o l'SDK hanno spesso decine o centinaia di pacchetti inclusi, potrebbe essere abbastanza difficile tenere traccia del contenuto del pacchetto. Lo strumento Poky responsabile di aiutarci in questo compito è la Build History.

La Build History, come si può intuire dal nome, conserva una cronologia dei contenuti di diversi manufatti costruiti durante l'uso Poky. Può tenere traccia della creazione del pacchetto, dell'immagine e dell'SDK.

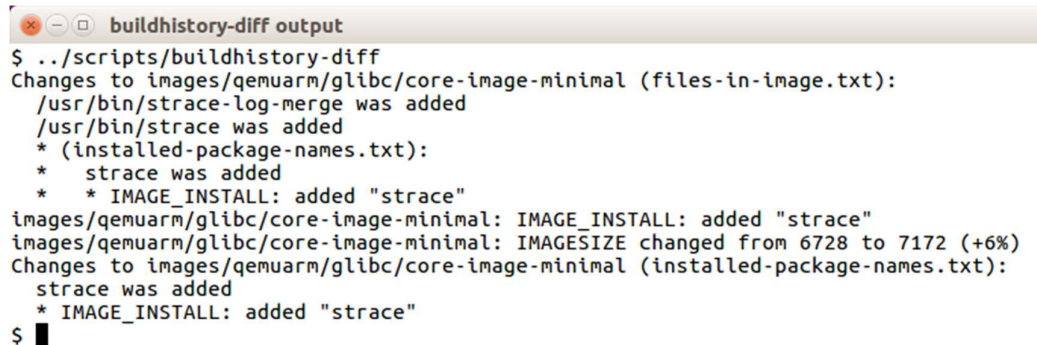
Per abilitare la Build History nel sistema, si devono aggiungere le seguenti righe di codice nel file `build/conf/local.conf`:

```
INHERIT += "buildhistory"  
BUILDHISTORY_COMMIT = "1"
```

Il metodo `INHERIT` include gli hook della classe `buildhistory` nel processo di build, mentre la riga `BUILDHISTORY_COMMIT` consente a BitBake di creare un nuovo commit Git nel repository `buildhistory` per ogni nuovo pacchetto, immagine o build SDK. Questo semplifica il tracciamento come un `git diff` tra due commit.

I dati per tutti i pacchetti, le immagini e gli SDK creati vengono archiviati nella directory `build/buildhistory` come file di testo in modo che sia facile utilizzarli per estrarre informazioni aggiuntive. Poky fornisce un'utilità che restituisce la differenza tra due stati `buildhistory`, chiamata `buildhistory-diff`, in un modo più conciso, che è molto utile quando si controllano le modifiche.

L'utilità `buildhistory-diff` restituisce la differenza tra due revisioni Git qualsiasi in un modo più significativo. Possiamo vedere un esempio del suo output nello screenshot seguente:



```

$ ../scripts/buildhistory-diff
Changes to images/qemuarm/glibc/core-image-minimal (files-in-image.txt):
  /usr/bin/strace-log-merge was added
  /usr/bin/strace was added
  * (installed-package-names.txt):
  *   strace was added
  *   * IMAGE_INSTALL: added "strace"
images/qemuarm/glibc/core-image-minimal: IMAGE_INSTALL: added "strace"
images/qemuarm/glibc/core-image-minimal: IMAGE_SIZE changed from 6728 to 7172 (+6%)
Changes to images/qemuarm/glibc/core-image-minimal (installed-package-names.txt):
  strace was added
  * IMAGE_INSTALL: added "strace"
$

```

Lo screenshot precedente mostra le differenze evidenziate da `buildhistory-diff` quando il pacchetto `strace` viene aggiunto nell'immagine `core-image-minimal`.

Quando viene eseguito il build di un pacchetto, `buildhistory` crea un elenco di sotto-pacchetti generati, script di installazione, un elenco di proprietari e dimensioni dei file, la relazione di dipendenza e altro ancora. Per le immagini e gli SDK, viene creata la relazione di dipendenza tra i pacchetti, i file del filesystem e il grafico delle dipendenze.

Per una migliore comprensione di tutte le capacità e caratteristiche fornite da `buildhistory`, si consiglia di leggere il manuale in <http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html>.

## Debugging dei pacchetti

Nelle ricette più sofisticate, si dividono i contenuti installati in diversi sotto-pacchetti. I pacchetti secondari possono essere funzioni, moduli o qualsiasi altro set di file è facoltativo installare.

Per controllare come è stato suddiviso il contenuto della ricetta, possiamo usare la directory `build/tmp/work/<arch>/<recipe name>/<software version>/packages-split`. Contiene una sottodirectory per ogni sotto-pacchetto e ha il suo contenuto nel sotto-albero.

Tra i possibili motivi per un'errata suddivisione dei contenuti, abbiamo i seguenti:

- Contenuto non installato (ad esempio, un errore negli script di installazione)
- Errore di configurazione dell'applicazione o della libreria (ad esempio, una funzione disabilitata)
- Errore nei metadati (ad esempio, ordine del pacchetto errato)

Un altro problema comune che troviamo, principalmente nelle ricette delle librerie, è che gli artefatti richiesti non vengono resi disponibili nella directory `sysroot` (ad esempio, header o librerie dinamiche), causando un'interruzione della build. La controparte della generazione `sysroot` può essere vista in `build/tmp/work/<arch>/<recipe name>/<software version>/sysroot-destdir`.

Altre possibili cause potrebbero richiedere la strumentazione del codice dei task con le ulteriori funzioni di log, in modo da poter capire l'errore logico o il bug che causa il risultato imprevisto.

## Le informazioni dei log durante l'esecuzione dei task

Le utilità di log fornite da BitBake sono molto utili per il tracing del path di esecuzione del codice. BitBake fornisce funzioni di log da utilizzare nel codice Python e Shell Script, descritte come segue:

- **Python:** Per l'uso all'interno delle funzioni Python, BitBake supporta diversi livelli di log, che sono `bb.fatal`, `bb.error`, `bb.warn`, `bb.note`, `bb.plain` e `bb.debug`
- **Shell Script:** Per l'uso nelle funzioni dello script della shell, esiste lo stesso insieme di livelli di log e si accede con una sintassi simile: `bbfatal`, `bberror`, `bbwarn`, `bbnote`, `bbplain` e `bbdebug`

Queste funzioni di log sono molto simili tra loro ma presentano differenze interne, descritte come segue:

- **`bb.fatal` e `bbfatal`:** Hanno la massima priorità di logging dei messaggi mentre stampano il messaggio e terminano l'elaborazione. Causano l'interruzione della build.
- **`bb.error` e `bberror`:** Vengono utilizzati per visualizzare un errore ma non forzano l'arresto della build.
- **`bb.warn` e `bbwarn`:** Vengono utilizzati per avvisare gli utenti di qualcosa.
- **`bb.note` e `bbnote`:** Aggiungono una nota per l'utente. Sono solo informative.
- **`bb.plain` e `bbplain`:** Generano un messaggio.
- **`bb.debug` e `bbdebug`:** Aggiungono informazioni a seconda del livello di debug utilizzato.

C'è una sottile differenza tra l'uso delle funzioni di log in Python e nello Shell Scripting.

Le funzioni di log in Python sono gestite direttamente da BitBake, visualizzate sulla console e memorizzate nel log di esecuzione visualizzabile all'interno di `build/tmp/log/cooker/<machine>`.

Quando le funzioni di log vengono utilizzate in Shell Script, le informazioni vengono inviate al rispettivo file di log del task, disponibile in `build/tmp/work/<arch>/<recipe name>/<software version>/temp`.

Al suo interno, possiamo eseguire gli script per ogni task con il pattern `run.<task>.<pid>` e utilizzare il pattern `log.<task>.<pid>` per il suo output. Per comodità, i link simbolici vengono mantenuti aggiornati da BitBake, puntando agli ultimi file di log utilizzando il pattern `log.<task>`; quindi, possiamo effettivamente verificare `log.do_compile`, ad esempio, quando intendiamo verificare se sono stati utilizzati i file corretti durante il processo di build. In alternativa, possiamo controllare `log.do_patch` per verificare se è stata applicata una patch.

La directory `build/tmp/work` viene descritta meglio nel [Capitolo 5](#), Dettagli sulla Directory Temporanea di Build.

## Utilizzo di una shell di sviluppo

Quando si modificano pacchetti o si esegue il debug di errori di build, può essere uno strumento utile una shell di sviluppo. Quando utilizziamo `devshell`, i file sorgenti vengono estratti nella directory di lavoro, vengono applicate le patch, viene aperto un nuovo terminale e i file vengono inseriti nella directory di lavoro.

Nel nuovo terminale, tutte le variabili d'ambiente necessarie per la build sono ancora definite, quindi possiamo usare comandi come `configure` e `make`. I comandi vengono eseguiti proprio come se il sistema di compilazione li stesse eseguendo.

Il comando seguente è un esempio che utilizza `devshell` su un target denominato `linux-yocto`:

```
$ bitbake linux-yocto -c devshell
```

Questo ci permette di rielaborare il codice sorgente del kernel Linux e lanciare la build sul posto, per evitare una build da zero sulla macchina di sviluppo, e cambiarne il codice secondo necessità.

È importante tenere a mente che nessuna modifica apportata all'interno di devshell permane tra le build; quindi, dobbiamo fare attenzione a registrare qualsiasi cambiamento importante, prima di uscire.

Poiché abbiamo il sorgente a nostra disposizione, possiamo usarlo per generare patch extra. Un modo molto pratico per farlo è usare Git e `git format-patch` per creare la patch da includere in seguito nella ricetta. Lo screenshot seguente mostra la finestra devshell aperta dopo aver chiamato il task `devshell`:

```

x - devshell example
$ bitbake linux-yocto -c devshell
Loading cache: 100% |#####| Time: 0:00:00
Loaded 1275 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "1.35.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "universal"
TARGET_SYS      = "arm-poky-linux-gnueabi"
MACHINE         = "qemuarm"
DISTRO          = "poky"
DISTRO_VERSION  = "2.4"
TUNE_FEATURES   = "arm armv5 thumb dsp"
TARGET_FPU      = "soft"
meta
meta-poky
meta-yocto-bsp   = "rocko:65d23bd7986615fdfb0f1717b615534a2a14ab80"

Initialising tasks: 100% |#####|
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
Currently 1 running tasks (286 of 286)
0: linux-yocto-4.12.12+gitAUTOINC+eda4d18
$ ls
CREDITS  firmware  ipc      lib      net      security  virt
block    crypto    fs       Kbuid    README  sound
certs    Documentation  include  Kconfig  Makefile  samples  tools
COPYING  drivers   init     kernel   mm        scripts  usr
$ export | grep ARCH
declare -x ARCH="arm"
declare -x UBOOT_ARCH="arm"
$

```

La `devshell` è conveniente per piccoli task, ma quando è necessaria una modifica più complessa, si usa una toolchain esterna in un normale ciclo di sviluppo per poi integrare le patch risultanti nella ricetta.

Per includere la patch generata nella ricetta e renderla persistente, vedere il [Capitolo 12, Personalizzazione delle ricette esistenti](#).

## Utilizzo di GNU Project Debugger per il debug

Durante lo sviluppo di qualsiasi progetto, di tanto in tanto finiamo per faticare a comprendere subdoli bug. Lo GNU Project Debugger (GDB) è disponibile come pacchetto all'interno di Poky ed è installato nelle immagini SDK per default, come descritto in dettaglio nel [Capitolo 8, Sviluppo con lo Yocto Project](#).

*Per installare pacchetti di debug che contengono i simboli e i tool di debug in un'immagine, si aggiunge `IMAGE_FEATURES += "dbg-pkgs tools-debug"` in `build/conf/local.conf`.*

L'uso dell'SDK, o un'immagine con i pacchetti e i tool di debug installati, consente di eseguire il debug delle applicazioni direttamente nel target, come di solito facciamo sulla macchina di sviluppo.

Il GDB potrebbe non essere utilizzabile su alcuni target a causa di limiti di memoria o spazio su disco. Il motivo principale di questa limitazione è che il GDB deve caricare le informazioni di debug, oltre ai file binari del processo in fase di debug, prima di avviare il processo di debug.

Per superare questi vincoli, si può utilizzare `gdbserver`, incluso per default quando si utilizza `tools-debug` in `IMAGE_FEATURES`. Viene eseguito sul target e non carica alcuna

informazione di debug dal processo sottoposto a debug. Al contrario, un'istanza GDB elabora le informazioni di debug sull'host. L'host GDB invia poi i comandi di controllo a `gdbserver` per controllare l'applicazione in debug.

Poiché l'host GDB è responsabile del caricamento delle informazioni di debug e dell'elaborazione necessaria per eseguire il processo di debug, sul target non è necessario che siano installati i simboli di debug e dobbiamo assicurarci che l'host possa accedere ai binari con le loro informazioni di debug. È consigliabile che i binari del target siano compilati senza ottimizzazioni per facilitare il processo di debug.

Il processo per utilizzare `gdbserver` e configurare l'host e il target nel modo appropriato è dettagliato sul seguente sito: <http://www.yoctoproject.org/docs/2.4/dev-manual/dev-manual.html>.

## Sommario

In questo capitolo abbiamo capito come possiamo configurare Poky per il processo di debug. Abbiamo visto il contenuto delle directory distribuite che utilizzabili per il debug, come tenere traccia delle modifiche usando `buildhistory`, come usare `devshell` per emulare lo stesso ambiente di build trovato da BitBake e come configurare il sistema per fornire gli strumenti necessari per il debug con GDB.

Nel prossimo capitolo capiremo come espandere il codice sorgente di Poky usando layer esterni. Verremo introdotti al concetto di layering [stratificazione] e impareremo in dettaglio la struttura delle directory e il contenuto di ciascun tipo di layer.

## 10. Esplorazione dei Layer Esterni

Una delle caratteristiche più affascinanti di Poky è la flessibilità nell'utilizzo dei layer esterni. In questo capitolo, esamineremo perché questa è una capacità forte e come possiamo trarne vantaggio. Analizzeremo i diversi tipi di layer e come sono organizzate le directory. Infine, impareremo come includere un nuovo layer nel progetto.

### *Potenziare la flessibilità con i layer*

Poky contiene una grande quantità di metadati sparsi su file di definizione macchina, classi e ricette. Questi metadati coprono tutto, dalle semplici applicazioni agli stack grafici a completi framework. BitBake ha la capacità di caricare metadati da più posizioni e questi set di metadati multipli sono noti come layer di metadati.

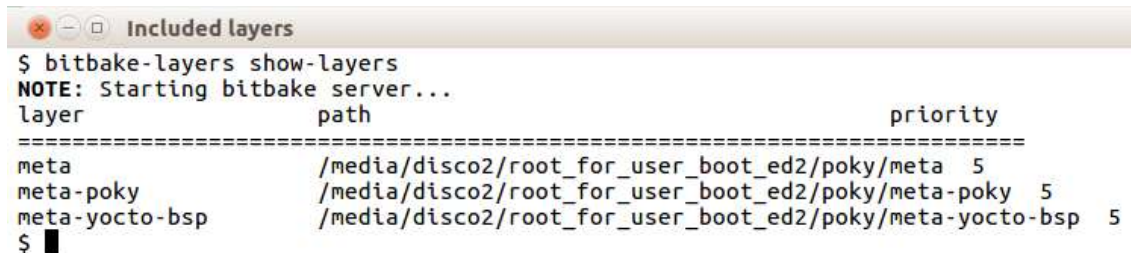
Il più grande punto di forza dell'utilizzo dei layer è la capacità di dividere i metadati in unità logiche, consentendo agli utenti di selezionare solo il set di metadati necessario per il progetto. L'uso dei layer di metadati migliora il riutilizzo del codice e la capacità di condividere il lavoro tra diversi team, comunità e fornitori, aumentando la qualità del codice della comunità dello Yocto Project poiché più entità lavorano insieme sugli stessi metadati.

Potremmo voler configurare il sistema per diversi motivi, come la necessità di abilitare/disabilitare una funzionalità o modificare i flag di build per abilitare ottimizzazioni specifiche dell'architettura. Questi sono esempi di personalizzazioni che possono essere eseguite utilizzando i layer.

Inoltre, quando creiamo il nostro ambiente di progetto personalizzato, invece di modificare ricette e file di configurazione e modificare i file nel layer Poky, dovremmo organizzare i metadati in layer diversi. Più l'organizzazione è separata, più facile sarà riutilizzare i layer nei progetti futuri; per questo motivo, anche il codice sorgente Poky stesso è suddiviso in diversi layer. Ha tre layer inclusi per default, come possiamo vedere nell'output della seguente riga di comando:

```
$ bitbake-layers show-layers
```

Il risultato è visibile nella seguente schermata:



```

$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                                path                                priority
=====
meta                                /media/disco2/root_for_user_boot_ed2/poky/meta  5
meta-poky                           /media/disco2/root_for_user_boot_ed2/poky/meta-poky  5
meta-yocto-bsp                       /media/disco2/root_for_user_boot_ed2/poky/meta-yocto-bsp  5
$
```

L'output della riga di comando mostra le seguenti tre importanti proprietà di qualsiasi layer:

- **Name:** Solitamente inizia con la stringa **meta**.
- **Path:** Questo è importante quando si vuole aggiungere un ulteriore layer nel progetto che viene aggiunto alla variabile **BBPATH**.
- **Priority:** Questo è il valore utilizzato da BitBake per decidere quale ricetta utilizzare e l'ordine in cui i file **.bbappend** devono essere riuniti. Significa che se due layer includono lo stesso file di ricetta (**.bb**), viene utilizzato quello con la priorità più alta. Nel caso di **.bbappen**, ogni file **.bbappend** è incluso nella ricetta originale e la priorità del layer determina l'ordine di inclusione, quindi i file **.bbappend** all'interno dei layer con priorità più alta vengono aggiunti per primi, seguiti dagli altri.

Poky è organizzato in tre singoli layer, casualmente i tre tipi disponibili. Il **meta** layer sono i metadati OpenEmbedded Core, che contengono le ricette, le classi e i file di configurazione della macchina QEMU. È un layer software.

Un layer software include solo applicazioni o file di configurazione per le applicazioni e può essere utilizzato su qualsiasi architettura. C'è un enorme elenco di layer software. Per citarne solo alcuni, abbiamo **meta-java**, **meta-qt5** e **meta-browser**. Il **meta-java** layer fornisce il supporto per il runtime Java e per l'SDK, il **meta-qt5** layer include il supporto per Qt5 e **meta-browser** supporta diversi browser web come Firefox e Chrome.

Il layer **meta-yocto-bsp** è il riferimento Poky per il layer del `board support package` (BSP). Contiene i file di configurazione della macchina e le ricette per configurare i pacchetti per le macchine. Poiché è un livello BSP di riferimento, può essere utilizzato come esempio.

Il layer **meta-poky** è il riferimento Poky per il layer della distribuzione.. Contiene una configurazione di distribuzione utilizzata in Poky per default; questo è un esempio di un file di distribuzione. Questa distribuzione di default è descritta nel file `poky.conf` ed è ampiamente utilizzata per testare i prodotti. Tuttavia, a volte il prodotto potrebbe avere esigenze speciali e le modifiche in `build/conf/local.conf` dovranno essere apportate come richiesto.

Il file `build/conf/local.conf` è un file volatile che non dovrebbe essere tracciato da Git.

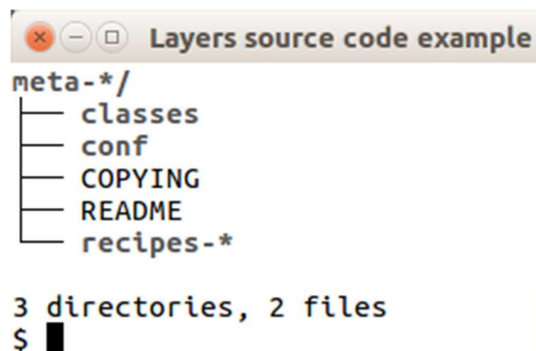
Non dovremmo fare affidamento su di esso per impostare le versioni dei pacchetti, i provider e le funzionalità di sistema per i prodotti, ma utilizzarlo solo come scorciatoia a scopo di test durante lo sviluppo.

La soluzione più adeguata e gestibile è creare un layer di distribuzione per posizionare il file di definizione della distribuzione. Questa configurazione consente di riprodurre in seguito qualsiasi build. Possiamo usare un layer di distribuzione per tutte le distribuzioni che abbiamo, oppure possiamo creare un nuovo layer per ogni nuova distribuzione; l'approccio migliore dipende dalle esigenze del progetto.

*La configurazione dei criteri fornita in un layer di distribuzione override [sovrascrive] la stessa configurazione da `build/conf/local.conf`.*

## Dettagliare il codice sorgente del layer

Di solito, un layer ha un'alberatura di directory, come mostrato nella schermata seguente:



```

Layers source code example
meta-*/
├── classes
├── conf
├── COPYING
├── README
└── recipes-*

3 directories, 2 files
$

```

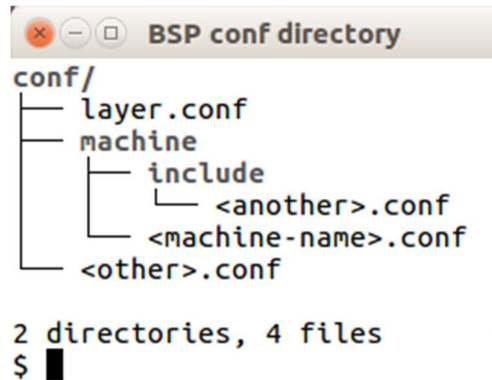
Il nome del layer dovrebbe iniziare con **meta-**; non è un requisito, ma la convenzione è consigliata. All'interno di questa directory ci sono due file, `<layer>/COPYING` e `<layer>/README`, una licenza e un messaggio per l'utente. In `<layer>/README`, dobbiamo specificare qualsiasi altra dipendenza e informazione che gli utenti del layer devono conoscere.

La cartella `classes` dovrebbe contenere sia le classi fornite che quelle specifiche per quel layer (i file `.bbclass`). È una directory opzionale.



La cartella `<layer>/conf` è obbligatoria e dovrebbe fornire i file di configurazione (i file `.conf`). In primo luogo, il file di configurazione del layer `<layer>/conf/layer.conf`, che verrà descritto in dettaglio nel prossimo capitolo, è il file con la definizione del layer.

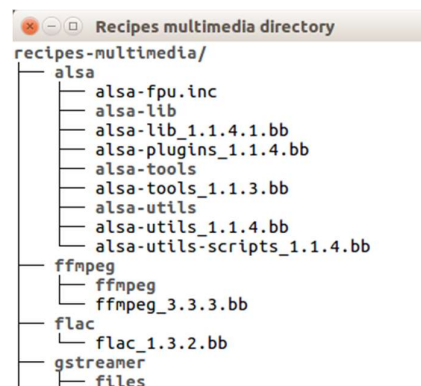
Quando la cartella `<layer>/conf` proviene da un layer BSP, la struttura della directory dovrebbe essere simile alla seguente schermata:



Se la cartella `<layer>/conf` proviene da un layer di distribuzione, la struttura della directory dovrebbe essere simile alla seguente schermata:



La cartella `recipe-*` è un gruppo di ricette separate per categoria, ad esempio, `recipes-core`, `recipes-bsp`, `recipes-graphic`, `recipes-multimedia` e `recipes-kernel`. All'interno di ogni cartella, partendo da `recipes-`, c'è una directory con il nome della ricetta o un gruppo di ricette; al suo interno, i file delle ricette terminano con `.bb` o `.bbappend`. Ad esempio, possiamo trovare il seguente screenshot da `meta`:



## Aggiunta di meta layer

Esistono centinaia di meta layer Yocto Project, OpenEmbedded, comunità e aziende che dovrebbero essere clonati manualmente all'interno della directory dei sorgenti del progetto per essere utilizzati. Si possono trovare su <http://git.yoctoproject.org/> o su <http://layers.openembedded.org>.

Per includere, ad esempio, `meta-openembedded` nel progetto, si può modificare il contenuto dei file di configurazione o utilizzare le righe di comando di BitBake. Per fare ciò, si deve prima recuperare il codice sorgente del layer. Eseguire il seguente comando dalla directory dei sorgenti di Poky:

```
$ git clone git://git.openembedded.org/meta-openembedded -b rocko
```

Ora possiamo aggiungere un nuovo layer modificando il file `build/conf/bblayers.conf` ed aggiungendo il path assoluto alla nuova directory del meta layer, come mostrato nel codice sorgente seguente. La riga evidenziata è quella da aggiungere. Le altre sono i valori di default per questo file:

```
build/conf/bblayers.conf content
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

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

BBLAYERS ?= " \
    /home/user/poky/meta \
    /home/user/poky/meta-poky \
    /home/user/poky/meta-yocto-bsp \
    "

BBLAYERS += \
    /home/user/poky/meta-openembedded/meta-oe \
    "
$
```

Un'alternativa alla modifica manuale di `build/conf/bblayers.conf` consiste nell'usare il tool `bitbake-layers` che esegue l'inclusione. Questo può essere fatto usando il seguente comando dalla directory build:

```
$ bitbake-layers add-layer ../meta-openembedded/meta-oe
```

Nel precedente comando BitBake, il layer aggiunto viene analizzato e il metadato `meta-openembedded/meta-oe` viene incluso nel database di BitBake, consentendo l'uso dei pacchetti all'interno del layer aggiunto.

## L'ecosistema di layer dello Yocto Project

Data la comodità di creare un layer, è possibile utilizzarne un numero enorme. Per facilitare l'accesso a tutti i livelli disponibili, la comunità di OpenEmbedded ha sviluppato un indice in cui è possibile trovarne la maggior parte.

Diciamo che abbiamo appena ordinato una scheda Raspberry Pi 3; possiamo usare il link a <https://layers.openembedded.org> e cercarlo nel tab Machines, come mostrato nella schermata seguente:

Branch: master ▾   Layers   Recipes   Machines   Distros		
raspberrypi		search
Machine name	Description	Layer
raspberrypi	RaspberryPI <a href="http://www.raspberrypi.org/">http://www.raspberrypi.org/</a> Board	<a href="#">meta-raspberrypi</a>
raspberrypi-cm	RaspberryPI Compute Module (CM1)	<a href="#">meta-raspberrypi</a>
raspberrypi-cm3	RaspberryPI Compute Module 3 (CM3)	<a href="#">meta-raspberrypi</a>
raspberrypi0	RaspberryPI Zero board ( <a href="https://www.raspberrypi.org/blog/raspberry-pi-zero">https://www.raspberrypi.org/blog/raspberry-pi-zero</a> )	<a href="#">meta-raspberrypi</a>
raspberrypi0-wifi	RaspberryPI Zero WiFi board ( <a href="https://www.raspberrypi.org/blog/raspberry-pi-zero-w-joins-family/">https://www.raspberrypi.org/blog/raspberry-pi-zero-w-joins-family/</a> )	<a href="#">meta-raspberrypi</a>
raspberrypi2	RaspberryPI 2	<a href="#">meta-raspberrypi</a>
raspberrypi3	RaspberryPI 3	<a href="#">meta-raspberrypi</a>
raspberrypi3-64	RaspberryPI 3 in 64 bits mode	<a href="#">meta-raspberrypi</a>

Un altro caso d'uso molto utile per gli strumenti è la ricerca di un tipo di software specifico o di una ricetta. Può salvare la giornata! Possiamo vedere alcuni layer comunemente usati nello screenshot seguente:

Branch: rocko ▾
Layers
Recipes
Machines
Distros

Search layers
Filter layers ▾

Layer name	Description	Type	Repository
<a href="#">meta-gnome</a>	GNOME UI support	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-networking</a>	Network-related software	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-android</a>	Android specific tools	Software	git://github.com/shr-distribution/meta-smartphone.git
<a href="#">meta-fsfilesystems</a>	Support for additional filesystems	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-initramfs</a>	Initramfs tools	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-luneui</a>	Recipes for LuneOS UI	Software	https://github.com/webOS-ports/meta-webos-ports
<a href="#">meta-mono</a>	Mono	Software	git://git.yoctoproject.org/meta-mono
<a href="#">meta-multimedia</a>	Multimedia-related software	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-perl</a>	Additional Perl recipes	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-python</a>	Python support	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-qt5</a>	Qt5 modules	Software	git://github.com/meta-qt5/meta-qt5.git
<a href="#">meta-webserver</a>	Web server related software	Software	git://git.openembedded.org/meta-openembedded
<a href="#">meta-xfce</a>	XFCE UI support	Software	git://git.openembedded.org/meta-openembedded

## Sommario

In questo capitolo abbiamo introdotto il concetto di layering [stratificazione]. Abbiamo appreso in dettaglio la struttura delle directory e il contenuto di ciascun tipo di layer. Inoltre, abbiamo visto come aggiungere manualmente un layer esterno al progetto o utilizzando la riga di comando di BitBake, nonché come utilizzare l'indice OpenEmbedded Layer per trovare facilmente i layer disponibili di cui abbiamo bisogno.

Nel prossimo capitolo impareremo di più sul perché è necessario creare nuovi layer e quali sono i metadati comuni in essi inclusi (come file di definizione macchina, ricette e immagini) e concluderemo il tutto con un esempio di personalizzazione della distribuzione.

# 11. Creazione di Layer Personalizzati

Oltre a utilizzare i layer esistenti della community o dei fornitori, in questo capitolo impariamo perché creiamo layer per i nostri prodotti. Inoltre, scopriamo come creare una definizione di macchina e una distribuzione, e ne approfittiamo per organizzare al meglio il codice sorgente.

## Creare un nuovo layer

Prima di creare il nostro layer, è sempre una buona idea verificare se ce n'è uno simile già disponibile sul seguente sito Web: <http://layers.openembedded.org>. Se non riusciamo a trovare un layer adatto per le nostre esigenze, il passo successivo è creare la directory. Di solito, il nome del layer inizia con `meta-`, ma questa non è una restrizione tecnica.

Il file di configurazione del layer è richiesto in ogni layer e si trova in `<layer>/conf/layer.conf`; possiamo crearlo manualmente utilizzando qualsiasi editor di testo, oppure popolarlo con uno script fornito in Poky, come mostrato nel comando seguente:

```
$ ./poky/scripts/yocto-layer create newlayer
```

L'output è mostrato nella schermata seguente:

```

Creating a new layer meta-newlayer
$ ./poky/scripts/yocto-layer create newlayer
Please enter the layer priority you'd like to use for the layer: [default: 6]
Would you like to have an example recipe created? (y/n) [default: n] y
Please enter the name you'd like to use for your example recipe: [default: example]
Would you like to have an example bbappend file created? (y/n) [default: n] y
Please enter the name you'd like to use for your bbappend file: [default: example]
Please enter the version number you'd like to use for your bbappend file (this should
match the recipe you're appending to): [default: 0.1]

New layer created in meta-newlayer.

Don't forget to add it to your BBLAYERS (for details see meta-newlayer/README).
$
```

Con lo script, ci viene chiesto di inserire il valore per la priorità del layer e rispondere ad altre domande relative al contenuto di esempio che può essere generato per il layer. Possiamo utilizzare i valori di default o inserirne di personalizzati. Un esempio di un layer generato è mostrato nella figura seguente:

```

Content of meta-newlayer
meta-newlayer/
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
├── recipes-example
│   └── example
│       ├── example-0.1
│       │   ├── example.patch
│       │   └── helloworld.c
│       └── example_0.1.bb
├── recipes-example-bbappend
│   └── example-bbappend
│       ├── example-0.1
│       │   ├── example.patch
│       └── example_0.1.bbappend
└── 7 directories, 8 files
$
```

Le variabili importanti che potrebbero dover essere aggiunte nel caso in cui il nostro layer richieda altri layer per funzionare, sono le seguenti:

- **LAYERVERSION:** Questa è una variabile facoltativa che specifica la versione del layer in un unico numero. Questa variabile viene utilizzata all'interno della variabile **LAYERDEPENDS** per dipendere da una versione specifica di un layer e deve avere come suffisso il nome del layer, per esempio, **LAYERVERSION\_newlayer** = "1".
- **LAYERDEPENDS:** Elenca i layer da cui dipendono le ricette, separati da spazi. Facoltativamente, possiamo assegnare una versione di layer specifica per una dipendenza aggiungendola alla fine del nome del layer con due punti, ad esempio, **otherlayer:2**. Questa variabile deve avere il suffisso del nome del layer specifico, ad esempio, **LAYERDEPENDS\_newlayer** = "otherlayer".

Se una dipendenza non può essere soddisfatta o i numeri di versione non corrispondono, viene generato un errore. La base della struttura del layer è ora creata. Nelle sezioni seguenti impareremo come estenderla.

## Aggiunta di metadati al layer

Lo scopo dietro l'uso dei layer è quello di aggiungere ulteriori metadati al database di BitBake o cambiarli.

Le funzionalità aggiunte più comunemente sono correlate al progetto, come applicazioni, librerie o un servizio server.

D'altra parte, invece di aggiungere nuove funzionalità, è molto più comune adattare le configurazioni delle funzionalità esistenti alle nostre esigenze, ad esempio i valori di rete iniziali per un server SSH o l'immagine 'splash' di boot personalizzata.

Significa che possiamo includere diversi tipi di file di metadati su un nuovo layer (ricette, immagini e file **bbappend**) per modificare le funzionalità esistenti. Lo script utilizzato per creare il nuovo layer può anche creare due file di esempio: il primo, **example\_0.1.bb**, è un esempio di ricetta; il secondo, **example\_0.1.bbappend**, è un esempio di **bbappend** utilizzato per modificare la funzionalità inclusa in **example\_0.1.bb**. Ci sono molti altri esempi di file **bbappend** su **meta-yocto-bsp** e **meta-yocto**, ed esploreremo alcuni dei loro usi comuni nel prossimo capitolo.

## Creazione di un'immagine

I file immagine possono essere visti come un insieme di pacchetti raggruppati per uno scopo e configurati in modo controllato. Possiamo creare una nuova immagine, inclusa un'immagine esistente, aggiungendo i pacchetti necessari o sovrascrivendo le configurazioni, oppure possiamo creare l'immagine da zero.

Quando un'immagine soddisfa per lo più le nostre esigenze e abbiamo bisogno di apportare solo piccole modifiche ad essa, è molto conveniente riutilizzarne il codice. Ciò semplifica la manutenzione del codice ed evidenzia le differenze funzionali. Ad esempio, se vogliamo includere un'applicazione e rimuovere una funzione dell'immagine da **core-image-sato**, possiamo creare un'immagine in **recipes-mine/images/my-image-sato.bb** con le seguenti righe di codice:

```
require recipes-sato/image/core-image-sato.bb
IMAGE_FEATURES_remove = "splash"
CORE_IMAGE_EXTRA_INSTALL += "myapp"
```

D'altra parte, a volte si vuol creare l'immagine da zero; possiamo facilitare il nostro lavoro usando la classe **core-image**, in quanto fornisce una serie di caratteristiche dell'immagine che possono essere utilizzate molto facilmente, ad esempio, un'immagine in **recipes-mine/images/myimage-nano.bb** è composta dalle seguenti righe di codice:

```
inherit core-image
IMAGE_FEATURES += "ssh-server-openssh splash"
CORE_IMAGE_EXTRA_INSTALL += "nano"
```

*L'operatore di accodamento (+=) viene usato per garantire che una nuova variabile IMAGE\_FEATURES possa essere aggiunta da build/conf/local.conf.*

**CORE\_IMAGE\_EXTRA\_INSTALL** è la variabile che dovremmo usare per includere pacchetti extra nell'immagine quando ereditiamo la classe **core-image** che facilita la creazione dell'immagine. Ciò aggiunge il supporto per la variabile **IMAGE\_FEATURES**, che evita molte duplicazioni del codice. La variabile **IMAGE\_INSTALL** raggruppa i contenuti **CORE\_IMAGE\_EXTRA\_INSTALL** e i pacchetti relativi a **IMAGE\_FEATURES** generano il filesystem di root.

Attualmente, le seguenti sono le funzioni supportate dell'immagine:

- **allow-empty-password**: Consente a Dropbear e OpenSSH di accettare accessi root e accessi da account con una stringa di password vuota
- **dbg-pkgs**: Installa i pacchetti di simboli di debug per tutti i pacchetti installati in una determinata immagine
- **debug-tweaks**: Rende un'immagine adatta per lo sviluppo
- **dev-pkgs**: Installa i pacchetti di sviluppo (header e link a librerie extra) per tutti i pacchetti installati in una determinata immagine
- **doc-pkgs**: Installa i pacchetti di documentazione per tutti i pacchetti installati in una determinata immagine
- **eclipse-debug**: Fornisce supporto per il debug remoto di Eclipse IDE
- **empty-root-password**: Imposta la password di root su una stringa vuota, che consente gli accessi con una password vuota
- **hwcodecs**: Installa i codec di accelerazione hardware
- **nfs-server**: Installa un server NFS
- **package-management**: Installa gli strumenti di gestione dei pacchetti e conserva il database del gestore dei pacchetti
- **perf**: Installa strumenti di profiling come **perf**, **systemtap** e **LTTng**
- **post-install-logging**: Abilita il log degli script post-installazione nel file `/var/log/postinstall.log` al primo boot dell'immagine sul sistema di target
- **pptest-pkgs**: Installa i pacchetti **pptest** per tutte le ricette abilitate per **pptest**
- **read-only-rootfs**: Crea un'immagine il cui filesystem di root è di sola lettura
- **splash**: Consente di mostrare una schermata iniziale durante il boot
- **ssh-server-dropbear**: Installa il server SSH minimo di Dropbear
- **ssh-server-openssh**: Installa il server OpenSSH, che è più completo di Dropbear
- **staticdev-pkgs**: Installa pacchetti di sviluppo statici, che sono librerie statiche (ovvero file \*.a) per tutti i pacchetti installati in una determinata immagine
- **tools-debug**: Installa strumenti di debug come **strace** e **gdb**
- **tools-sdk**: Installa un SDK completo che viene eseguito sul dispositivo
- **tools-testapps**: Installa strumenti di test del dispositivo (ad esempio, debug del touchscreen)
- **x11**: Installa il server X
- **x11-base**: Installa il server X con un ambiente minimo
- **x11-sato**: Installa l'ambiente OpenedHand Sato

## Aggiunta di una ricetta del pacchetto

Una ricetta del pacchetto è il modo in cui possiamo istruire BitBake a recuperare, decomprimere, compilare e installare la nostra applicazione, il modulo del kernel o

qualsiasi software fornito da un progetto. Poky include diverse classi che astraggono il processo per gli strumenti di sviluppo più comuni come quelli basati su Autotools, CMake e QMake. Un elenco delle classi incluse in Poky può essere visto nel seguente manuale di riferimento: <http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html>.

Una semplice ricetta che esegue le attività di compilazione e installazione in modo esplicito viene fornita come segue:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0b
cf8506ecda2f7b4f302"
SRC_URI = "file://helloworld.c"
S = "${WORKDIR}"
do_compile() {
    ${CC} helloworld.c -o helloworld
}
do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

I blocchi di `do_compile` e `do_install` forniscono i comandi Shell Scripting per creare e installare il binario risultante nella directory di destinazione a cui si fa riferimento come `${D}`.

Tuttavia, nel caso di un progetto basato su Autotools, possiamo evitare molte duplicazioni di codice usando la classe `autotools` nell'esempio estratto dalla ricetta `poky/meta/recipes-core/dbus-wait/dbus-wait_git.bb`, nel modo seguente:

```
DESCRIPTION = "A simple tool to wait for a specific signal over DBus"
...
inherit autotools
```

Il semplice atto di ereditare la classe è, infatti, fornire tutto il codice necessario per svolgere i seguenti task:

- Aggiornare il codice dello script di configurazione e gli artefatti
- Aggiornare gli script di libtool
- Eseguire lo script di configurazione
- Eseguire Make
- Eseguire Make install

Gli stessi concetti si applicano ad altri strumenti di build, come nel caso di CMake e QMake. Il numero di classi supportate sta crescendo e si prevede che ne verranno incluse di nuove in ogni versione per supportare i nuovi sistemi di build ed evitare la duplicazione del codice.

## Creazione automatica di una ricetta del pacchetto base utilizzando "recipetool"

Il tool `recipetool` consente di creare più facilmente una ricetta di base basata sui file del codice sorgente. Finché è possibile estrarre o puntare ai file sorgenti, `recipetool` genererà una ricetta e configurerà automaticamente tutte le informazioni predefinite nel nuovo file.

Per illustrare, supponiamo di avere un'applicazione che compila utilizzando Autotools. Quando utilizziamo `recipetool` per creare la ricetta di base, esso genera una ricetta che ha le dipendenze di pre-compilazione, eredita la classe `autotools`, imposta i requisiti di licenza e checksum.

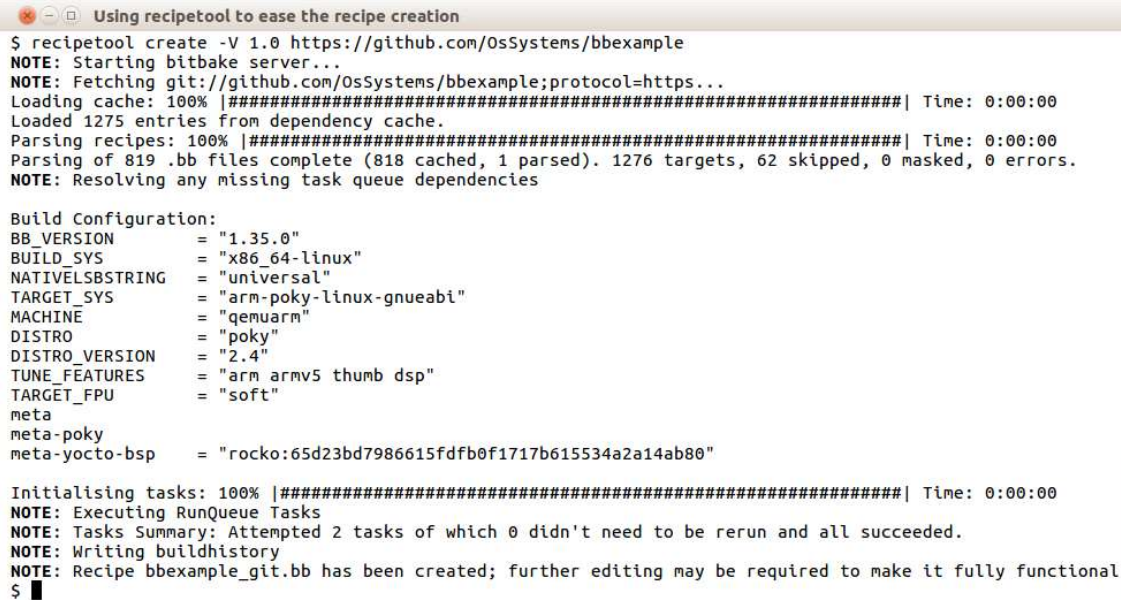
*Il tool recipetool crea una ricetta con diversi commenti destinati alla comprensione del contenuto. Tutti i commenti possono essere cancellati quando integriamo il file della ricetta nel nostro metalayer.*



Per generare una ricetta utilizzando `bbexample`, disponibile in <https://github.com/OSSystems/bbexample>, possiamo utilizzare i seguenti comandi:

```
$ source oe-init-build-env build
$ recipetool create -V 1.0 https://github.com/OSSystems/bbexample
```

L'immagine seguente mostra la creazione della ricetta:



```

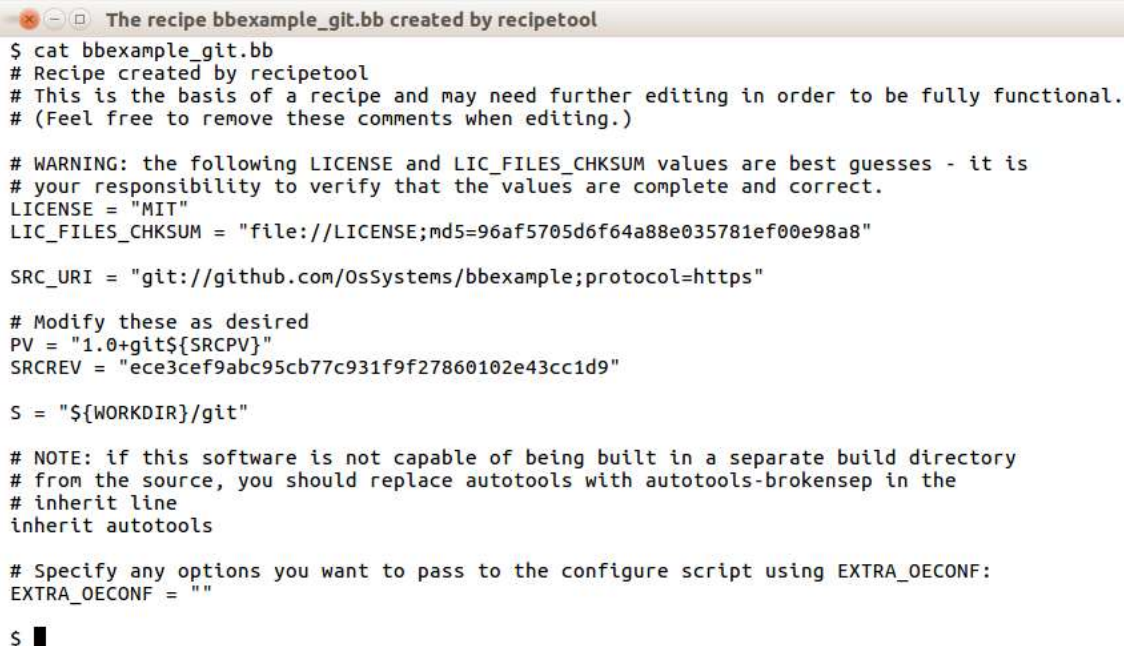
$ recipetool create -V 1.0 https://github.com/OSSystems/bbexample
NOTE: Starting bitbake server...
NOTE: Fetching git://github.com/OSSystems/bbexample;protocol=https...
Loading cache: 100% |#####| Time: 0:00:00
Loaded 1275 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:00
Parsing of 819 .bb files complete (818 cached, 1 parsed). 1276 targets, 62 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "1.35.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "universal"
TARGET_SYS      = "arm-poky-linux-gnueabi"
MACHINE         = "qemuarm"
DISTRO          = "poky"
DISTRO_VERSION  = "2.4"
TUNE_FEATURES   = "arm armv5 thumb dsp"
TARGET_FPU      = "soft"
meta
meta-poky
meta-yocto-bsp   = "rocko:65d23bd7986615fdb0f1717b615534a2a14ab80"

Initialising tasks: 100% |#####| Time: 0:00:00
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and all succeeded.
NOTE: Writing buildhistory
NOTE: Recipe bbexample_git.bb has been created; further editing may be required to make it fully functional
$

```

Il tool `recipetool` crea il file `bbexample_git.bb` dopo aver scaricato il codice sorgente dall'URL e analizzato il suo contenuto. In base al codice sorgente crea la base della ricetta, come mostrato nell'immagine seguente:



```

$ cat bbexample_git.bb
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing in order to be fully functional.
# (Feel free to remove these comments when editing.)

# WARNING: the following LICENSE and LIC_FILES_CHKSUM values are best guesses - it is
# your responsibility to verify that the values are complete and correct.
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=96af5705d6f64a88e035781ef00e98a8"

SRC_URI = "git://github.com/OSSystems/bbexample;protocol=https"

# Modify these as desired
PV = "1.0+git${SRCPV}"
SRCREV = "ece3cef9abc95cb77c931f9f27860102e43cc1d9"

S = "${WORKDIR}/git"

# NOTE: if this software is not capable of being built in a separate build directory
# from the source, you should replace autotools with autotools-brokensep in the
# inherit line
inherit autotools

# Specify any options you want to pass to the configure script using EXTRA_OECONF:
EXTRA_OECONF = ""

$

```

*Anche se `recipetool` crea una ricetta base, non dovrebbe essere considerata una ricetta finale. Si devono controllare le opzioni di compilazione, informazioni aggiuntive sui metadati e così via.*

Il file `bbexample_git.bb` viene creato nella directory in cui viene eseguito `recipetool` (`build`), dobbiamo quindi copiare il file nella posizione desiderata, nell'esempio `meta-`



`newlayer/recipes-mine/bbexample/bbexample_git.bb`. Dopo averlo copiato, possiamo eseguire il build usando BitBake come al solito.

## Aggiunta del supporto a una nuova definizione di macchina

Creare una nuova macchina che possa essere utilizzata da Poky è un compito semplice. Fornisce essenzialmente le informazioni necessarie per il funzionamento di una macchina. I driver del bootloader, del kernel e del supporto hardware devono essere controllati prima di iniziare a integrare la scheda nel layer BSP.

Lo Yocto Project supporta x86-32, x86-64, ARM32, ARM64, MIPS, MIPS64 e PowerPC, che rappresentano le architetture embedded attualmente più utilizzate.

L'insieme prevalente di variabili utilizzate in una definizione di macchina è il seguente:

- **TARGET\_ARCH**: Imposta l'architettura della macchina, ad esempio ARM o i586
- **PREFERRED\_PROVIDER\_virtual/kernel**: Sovrascrive il kernel di default (`linux-yocto`) nel caso in cui sia necessario utilizzarne uno specifico
- **SERIAL\_CONSOLES**: Definisce le console seriali e le loro velocità
- **MACHINE\_FEATURES**: Descrive le caratteristiche hardware, quindi lo stack software necessario viene incluso nelle immagini per default
- **KERNEL\_IMAGETYPE**: Viene utilizzato per scegliere il tipo di immagine del kernel, ad esempio `zImage` e `uImage`
- **IMAGE\_FSTYPES**: Imposta i tipi di immagine del filesystem generati, ad esempio, `tar.gz`, `ext4` e `ubifs`

Si possono vedere esempi di file di definizione della macchina all'interno del codice sorgente di Poky in `meta-yocto-bsp/conf/machine/`. Quando si descrive una nuova macchina, si dovrebbe prestare particolare attenzione alle funzionalità specifiche da essa supportate in **MACHINE\_FEATURES**. In questo modo, nelle immagini viene installato il software necessario per supportare tali funzionalità. I valori attualmente disponibili per **MACHINE\_FEATURES** sono elencati come segue:

- **acpi**: L'hardware ha l'ACPI (solo x86/x86\_64)
- **alsa**: L'hardware ha i driver audio ALSA
- **apm**: L'hardware usa APM (o l'emulazione APM)
- **bluetooth**: Hardware ha il BT integrato
- **efi**: Supporta il booting tramite l'EFI
- **ext2**: Hardware HDD o Microdrive
- **irda**: L'hardware ha il supporto per IrDA
- **keyboard**: L'hardware ha una tastiera
- **pcbios**: Supporto per il booting tramite il BIOS
- **pci**: L'hardware ha un bus PCI
- **pcmcia**: L'hardware ha connettori PCMCIA o CompactFlash
- **phone**: Supporto per il telefono cellulare (voce)
- **qvga**: La machine ha un display QVGA (320x240)
- **rtc**: La machine ha un clock real-time
- **screen**: L'hardware ha uno schermo
- **serial**: L'hardware ha il supporto per la seriale (solitamente RS232)
- **touchscreen**: L'hardware ha un touchscreen
- **usb gadget**: L'hardware è compatibile con i dispositivi gadget USB
- **usb host**: L'hardware è compatibile con host USB
- **vfat**: Supporto per il filesystem FAT

- **wifi**: L'hardware ha il Wi-Fi integrato

## Confezionare un'immagine per la macchina

Un aspetto che viene spesso trascurato fino alla fine dello sviluppo del layer di supporto BSP, è la creazione di un'immagine pronta per l'uso per la macchina. Il tipo di immagine che utilizzeremo dipende da molteplici aspetti: il processore, le periferiche incluse nella scheda e le limitazioni del progetto.

Il tipo di immagini più utilizzato per l'uso diretto nello storage [memoria] è la `partitioned image`. Lo Yocto Project ha un tool, chiamato `wic`, che fornisce un modo flessibile per generare tali immagini. Consente la creazione di immagini partizionate basate su un file template (`wks`) scritto in un linguaggio comune che descrive il layout dell'immagine del target. La definizione del linguaggio si trova nella documentazione:

<http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html#openembedded-kickstart-wks-reference>.

Il file `wks` viene inserito nel metalayer all'interno della directory `wic`. Non è raro avere in questa directory più file per specificare diversi layout di immagine; tuttavia, è importante tenere presente che il layout scelto deve corrispondere alla macchina.

Ad esempio, se si considera una macchina basata su i.MX che si avvia utilizzando SPL e U-Boot da una scheda SD con due partizioni, una per i file di avvio e l'altra per il `rootfs`. Le rispettive `wks` vengono mostrate qui:

```
# short-description: Create SD card image with a boot partition
# long-description:
# Create an image that can be written onto a SD card using dd for use
# with i.MX SoC family.
# It uses SPL and u-boot
#
# The disk layout used is:
# -----
# | | SPL | u-boot | boot | rootfs |
# -----
# ^ ^ ^ ^ ^
# | | | | |
# 0 1kiB 69kiB 4MiB 16MiB + rootfs + IMAGE_EXTRA_SPACE (default 10MiB)
#
part SPL --source rawcopy --sourceparams="file=SPL" --ondisk mmcblk --no-table --align 1
part u-boot --source rawcopy --sourceparams="file=u-boot.img" --ondisk mmcblk --no-table --align 69
part /boot --source bootimg-partition --ondisk mmcblk --fstype=vfat --label boot --active --align 4096 --size 16
part / --source rootfs --ondisk mmcblk --fstype=ext4 --label root --align 4096

bootloader --ptable msdos
```

Per abilitare la generazione di immagini basata su Wic, è necessario aggiungerla a **IMAGE\_FSTYPES**. Si possono anche definire il file `wks` da utilizzare impostando la variabile **WKS\_FILE**. Tali variabili sono dettagliate su <http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html>

## Utilizzo di una distribuzione personalizzata

La creazione di una distribuzione è un mix di semplicità e complessità. Il processo di creazione del file di distribuzione è molto semplice; tuttavia, la configurazione della distribuzione ha un forte impatto sul modo in cui Poky si comporta e può causare un'incompatibilità binaria con quelli precedentemente compilati, a seconda delle opzioni che utilizziamo.

La distribuzione è dove definiamo le opzioni globali, come la versione della toolchain, i backend grafici, il supporto per OpenGL e così via. Dovremmo effettuare una distribuzione solo nel caso in cui le impostazioni predefinite fornite da Poky non soddisfino i nostri requisiti.

Di solito, intendiamo cambiare una piccola serie di opzioni da Poky. Ad esempio, rimuoviamo il supporto X11 per utilizzare invece un framebuffer. Possiamo farlo facilmente riutilizzando la distribuzione Poky e sovrascrivendo le variabili di cui abbiamo bisogno. Ad

esempio, la distribuzione di esempio rappresentata dal file

`<layer>/conf/distro/mydistro.conf` è la seguente:

```
require conf/distro/poky.conf
DISTRO = "mydistro"
DISTRO_NAME = "mydistro (My New Distro)"
DISTRO_VERSION = "1.0"
DISTRO_CODENAME = "codename"
SDK_VENDOR = "-mydistrosdk"
SDK_VERSION := "${@}${DISTRO_VERSION}'.replace('snapshot-
${DATE}', 'snapshot')}"

MAINTAINER = "mydistro <mydistro@mycompany.com>"

DISTRO_FEATURES_remove = "x11"
```

Per utilizzare la distribuzione appena creata, si deve aggiungere il seguente pezzo di codice in `build/conf/local.conf`:

```
DISTRO = "mydistro"
```

La variabile `DISTRO_FEATURES` può influenzare la configurazione delle ricette e l'installazione dei pacchetti nelle immagini. Ad esempio, per essere in grado di utilizzare il suono in qualsiasi macchina e immagine, devono essere presenti le funzionalità `alsa`.

L'elenco seguente mostra lo stato attuale dei valori supportati da `DISTRO_FEATURES`:

- **alsa**: Include il supporto ALSA (moduli kernel di compatibilità OSS installati se disponibili)
- **api-documentation**: Abilita la generazione di documentazione delle API durante le build delle ricette
- **bluetooth**: Include il supporto Bluetooth (solo BT integrato)
- **bluez5**: Include BlueZ versione 5, che fornisce livelli Bluetooth di base e supporto dei protocolli
- **cramfs**: Include il supporto per CramFS
- **directfb**: Include il supporto per DirectFB
- **ext2**: Include strumenti per il supporto di dispositivi con HDD/Microdrive interni per l'archiviazione di file (anziché dispositivi solo con Flash)
- **ipsec**: Include il supporto per IPSec
- **ipv6**: Include il supporto per IPv6
- **irda**: Include il supporto per IrDA
- **keyboard**: Include il supporto della tastiera (ad esempio, le mappe dei tasti (Keymaps) verranno caricate durante il boot)
- **ldconfig**: Include il supporto per `ldconfig` e `ld.so.conf` sul target
- **nfs**: Include il supporto del client NFS (per gli export del montaggio NFS sul device)
- **opengl**: Include la `Open Graphics Library (OpenGL)`, che è un'interfaccia di programmazione di applicazioni cross-linguaggio, multi-piattaforma utilizzata per il rendering di grafica bi- e tri-dimensionale
- **pci**: Include il supporto per il bus PCI
- **pcmcia**: Include il supporto per PCMCIA/CompactFlash
- **ppp**: Include il supporto per la connessione remota PPP
- **ptest**: Abilita la creazione dei test dei pacchetti dove supportati dalle singole ricette
- **smbfs**: Include il supporto di client su reti SMB (per il montaggio di condivisioni Samba/Microsoft Windows sul device)
- **systemd**: Include il supporto per questo `init` manager, che è un sostituto completo di `sysvinit` con avvio parallelo dei servizi, un ridotto 'overhead' della shell e altre funzionalità

- **usb gadget:** Include il supporto del dispositivo gadget USB (per rete/seriale/archiviazione USB)
- **usb host:** Include il supporto per USB host (consente la connessione a tastiera, mouse, storage e rete esterni, tra gli altri)
- **wayland:** Include il protocollo del server di visualizzazione Wayland e la libreria che lo supporta
- **wifi:** Include il supporto per il Wi-Fi (solo integrato)
- **x11:** Include il server X e le librerie

## ***MACHINE\_FEATURES e DISTRO\_FEATURES***

Sia **DISTRO\_FEATURES** che **MACHINE\_FEATURES** lavorano insieme per fornire un supporto fattibile sul sistema finale.

Quando una macchina supporta una funzionalità, ciò non implica che sia supportata dal sistema finale perché la distribuzione utilizzata ne deve fornire la base sottostante.

Se una macchina supporta il Wi-Fi ma la distribuzione no, le applicazioni utilizzate dal sistema operativo verranno costruite con il supporto Wi-Fi disabilitato, in modo che il risultato sarà un sistema senza supporto Wi-Fi.

D'altra parte, se la distribuzione fornisce supporto Wi-Fi e una macchina no, i moduli e le applicazioni necessarie per il Wi-Fi non verranno installati nelle immagini create per questa macchina, sebbene il sistema operativo e i suoi moduli supportino l'abilitazione al Wi-Fi.

## ***Lo 'scope' delle variabili***

I metadati di BitBake hanno migliaia di variabili, ma lo scope in cui queste variabili sono disponibili dipende da dove vengono definite. Fondamentalmente, ci sono due tipi di variabili:

- Le variabili definite nei file di configurazione sono globali per ogni ricetta.  
L'ordine di analisi dei file di configurazione principali è mostrato di seguito:
  - `build/conf/local.conf`
  - `<layer>/conf/machines/<machine>.conf`
  - `<layer>/conf/distro/<distro>.conf`
- Le variabili definite all'interno della ricetta sono locali alla specifica ricetta solo durante l'esecuzione dei suoi task

## ***Sommario***

In questo capitolo abbiamo appreso le ragioni che ci motivano a creare un nuovo layer e nuovi metadati. Abbiamo visto una descrizione di come creare la configurazione della macchina, una definizione della distribuzione e le ricette. Abbiamo imparato come creare immagini e come includere la nostra applicazione in un'immagine.

Nel prossimo capitolo, accederemo ad alcuni esempi dei casi di personalizzazione più comuni utilizzati da un layer aggiuntivo, come la modifica di pacchetti esistenti, l'aggiunta di opzioni extra ad `autoconf`, l'applicazione di una nuova patch e l'inclusione di un nuovo file in un pacchetto. Vedremo come configurare `BusyBox` e `linux-yocto`, due pacchetti comunemente personalizzati quando si realizza un sistema embedded.

# 12. Personalizzazione di ricette esistenti

Nel corso del nostro lavoro con gli strumenti dello Yocto Project, ci si aspetta di dover personalizzare le ricette esistenti. In questo capitolo esploreremo alcuni esempi, come modificare le opzioni di compilazione, abilitare o disabilitare le funzionalità di una ricetta, applicare una patch aggiuntiva e modificare le impostazioni di `BusyBox` e di `Linux Yocto Framework`.

## Casi d'uso comuni

Al giorno d'oggi, i progetti di solito hanno una serie di layer per fornire le funzionalità richieste. Abbiamo sicuramente bisogno di apportare modifiche per adattarle alle esigenze specifiche. Possono essere modifiche estetiche o sostanziali, ma il modo per realizzarle è lo stesso.

Per apportare modifiche a una ricetta preesistente, si deve creare un file `.bbappend` nel layer del progetto. Il nome del file è lo stesso della ricetta originale, insieme al suffisso `append`. Ad esempio, se la ricetta originale si chiama `<original-layer>/recipes-core/app/app_1.0.bb`, il corrispondente `.bbappend` sarà `<layer>/recipes-core/app/app_1.0.bbappend`.

Il file `.bbappend` può essere visto come un pezzo di testo che viene aggiunto alla fine della ricetta originale. Ci fornisce un meccanismo estremamente flessibile per evitare la duplicazione del codice sorgente al fine di applicare le modifiche richieste ai layer del progetto.

Quando c'è più di un file `.bbappend` per una ricetta, vengono tutti uniti seguendo l'ordine di priorità del layer.

## Aggiunta di opzioni extra alle ricette basate su Autoconf

Supponiamo di avere l'applicazione basata sul sistema di build di Autoconf, insieme a una ricetta preesistente, e vogliamo fare quanto segue:

- Abilitare my-feature
- Disabilitare another-feature

Il contenuto del file `.bbappend`, per apportare le modifiche, sarà il seguente:

```
EXTRA_OECONF += "--enable-my-feature --disable-another-feature"
EXTRA_OEMAKE += "DEFINE_PASSED_TO_MAKE=1"
```

Questo può essere fatto anche in base all'hardware per cui stiamo eseguendo il build, come segue:

```
EXTRA_OECONF_append_arm = " --enable-my-arm-feature"
EXTRA_OEMAKE_append_mymachine = " MYMACHINE_SPECIFIC=1"
```

`EXTRA_OECONF` is used to add extra options to the configuration script.  
`EXTRA_OEMAKE` is used to add extra parameters to the make call.

## Applicazione di una patch

Per i casi in cui c'è bisogno di applicare una patch a un pacchetto esistente, si dovrebbe usare **FILESEXTRAPATHS**, che include nuove directory nell'algoritmo di ricerca, rendendo il file extra visibile a BitBake, come mostrato qui:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://mypatch.patch"
```

Nell'esempio precedente, **THISDIR** si espande alla directory corrente, e **PN** e **PV** si espandono rispettivamente al nome del pacchetto e alla sua versione. Questo nuovo path viene quindi incluso nell'elenco delle directory utilizzato per la ricerca dei file. L'uso dell'operatore **\_prepend** è importante in quanto garantisce l'utilizzo del file fornito, anche se in futuro verrà aggiunto un file con lo stesso nome nei layer di priorità inferiore.

BitBake presuppone che ogni file con estensione **.patch** sia una patch e le applica di conseguenza.

## Aggiunta di file extra ai pacchetti esistenti

Se dobbiamo includere un file di configurazione aggiuntivo, dovremmo usare **FILESEXTRAPATHS**, come spiegato nell'esempio precedente e mostrato nelle seguenti righe di codice:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://newconfigfile.conf"
do_install_append() {
    install -m 644 ${WORKDIR}/newconfig.conf ${D}${sysconfdir}
}
```

La funzione **do\_install\_append** accoda il blocco fornito sotto i metadati già disponibili nella funzione **do\_install** originale. Include il comando necessario per copiare il nuovo file di configurazione nel filesystem del pacchetto. Il file viene copiato da **\${WORKDIR}** a **\${D}** poiché queste sono le directory usate da Poky per il build del pacchetto e la directory di destinazione usata da Poky per creare il pacchetto. La directory **\${sysconfdir}** è la directory di configurazione del sistema (di solito con **/etc**).

Dovremmo usare le variabili fornite al di sopra di **poky/meta/conf/bitbake.conf**, invece di puntare a path 'hardcoded'. Ad esempio, usare **\${sysconfdir}** anziché **/etc**, e **\${bindir}** al posto di **/usr/bin**.

## I path di ricerca dei file

Quando un file (una patch o un file generico) viene incluso in **SRC\_URI**, BitBake cerca le variabili **FILESPATH** e **FILESEXTRAPATHS**. L'impostazione di default è quella di cercare nelle posizioni seguenti:

- **<recipe>-<version>/**
- **<recipe>/**
- **files/**

Oltre a ciò, verifica anche la presenza di un **OVERRIDES** per un file specifico da sovrascrivere in ciascuna cartella. Per illustrare ciò, si consideri una ricetta, **foo\_1.0.bb**, e la variabile **OVERRIDES = "<board>:<arch>"** il file verrà ricercato nelle seguenti directory, rispettando l'ordine esatto mostrato:

- **foo-1.0/<board>/**
- **foo-1.0/<arch>/**
- **foo-1.0/**
- **foo/<board>/**
- **foo/<arch>/**
- **foo/**
- **files/<board>/**

- files/<arch>/
- files/

Questo è solo illustrativo in quanto l'elenco di **OVERRIDES** è enorme e specifico per la macchina. Quando si con la ricetta, si può usare **bitbake -e** per scoprire l'elenco completo degli 'override' (sostituzioni) disponibili per una macchina specifica e usarle di conseguenza.

## Modifica della configurazione della funzione della ricetta

Un meccanismo supportato per semplificare la personalizzazione del set di funzionalità per le ricette è **PACKAGECONFIG**. Fornisce un modo per abilitare e disabilitare le funzioni della ricetta. Ad esempio, supponiamo che la ricetta abbia la seguente configurazione:

```
PACKAGECONFIG ?= "feature1"
PACKAGECONFIG[feature1] = "--enable-feature1,--disablefeature1, feature1depends"
PACKAGECONFIG[feature2] = "--enable-feature2,--disablefeature2, feature2depends"
```

La ricetta ha due funzionalità, **feature1** e **feature2**. Per ciascuna opzione di configurazione, è presente una stringa per definire come abilitare la funzione su **autoconf**, come disabilitare la funzione su **autoconf** e le nuove dipendenze nel caso in cui l'opzione sia abilitata.

Possiamo creare un file **.bbappend** che espande il valore di default della variabile **PACKAGECONFIG** per abilitare anche **feature2**, come mostrato qui:

```
PACKAGECONFIG += "feature2"
```

*Per aggiungere la stessa funzionalità al file **build/conf/local.conf**, possiamo utilizzare **PACKAGECONFIG\_pn-<recipename>\_append = 'feature2'**.*

Informazioni più dettagliate sull'uso di **PACKAGECONFIG** e le sue opzioni sono disponibili su <http://www.yoctoproject.org/docs/2.4/ref-manual/ref-manual.html>.

## Personalizzazione di BusyBox

BusyBox è un componente chiave della maggior parte dei progetti embedded basati su Linux in quanto fornisce un'alternativa alle utilità più comunemente utilizzate, ma con un ingombro inferiore rispetto alle sue solite controparti Linux. Può essere visto come una sorta di coltellino svizzero, poiché fornisce una vasta gamma di utilità ed è abbastanza flessibile riguardo a quali utilità abilitare o disabilitare.

Per deselezionare un'opzione, si può invece aggiungere una riga negativa, ad esempio, **CONFIG\_TFTPD=n**.

Poky fornisce un'impostazione di default per BusyBox e a volte potrebbe non soddisfare le nostre esigenze, quindi modificare questa configurazione è un'attività comune. Ad esempio, il file **<layer>/recipes-core/busybox/busybox\_%.bbappend** potrebbe avere le seguenti righe di codice:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://enable-tftpd.cfg"
```

Il file **<layer>/recipes-core/busybox/busybox/enable-tftpd.cfg** contiene quanto segue:

```
CONFIG_TFTPD=y
```

Questa combinazione del file **.bbappend** e del file di configurazione è sufficiente per abilitare il supporto per il server **TFTP** in BusyBox.

Quando un file **.bbappend** viene creato, l'operatore **%** viene usato come carattere jolly (come nell'esempio **<layer>/recipes-core/busybox/busybox\_%.bbappend**) e il file **.bbappend** verrà accodato a tutti i metadati della ricetta originale.

## Personalizzazione del framework linux-yocto

Il kernel Linux è un complesso software che fornisce un numero infinito di possibili configurazioni. Lo Yocto Project fornisce un framework (`linux-yocto`) per gestire un enorme insieme di macchine in un singolo albero del kernel. Possiamo sfruttare questo framework per abilitare o disabilitare le funzionalità per la macchina, ad esempio, usando `<layer>/recipes-kernel/linux/linux-yocto_%.bbappend` col contenuto seguente:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://enable-can.cfg"
```

Il contenuto del file `<layer>/recipes-kernel/linux/linux-yocto/linux-yocto/enable-can.cfg` è questo:

```
CONFIG_CAN=y
```

Un requisito comune durante la creazione di un sistema embedded basato su Linux è modificare la configurazione del kernel. Possiamo farlo usando l'SDK o BitBake, come spiegato qui:

- Con l'SDK: La creazione e l'installazione dell'SDK dello Yocto Project descritte in dettaglio nel [Capitolo 8, Sviluppo con lo Yocto Project](#). Dopo aver esportato l'SDK, possiamo configurare il sorgente del kernel Linux nel solito modo (ad esempio, `make menuconfig`).
- Col BitBake: Quando sono necessarie piccole modifiche o test, possiamo utilizzare BitBake per configurare o generare il file di configurazione del kernel Linux. Possiamo usare i seguenti comandi per raggiungere questo obiettivo:

```
$ bitbake virtual/kernel -c menuconfig
$ bitbake virtual/kernel -c savedefconfig
```

Per lo sviluppo del kernel Linux, l'uso dell'SDK è preferito in quanto fornisce un comodo ambiente di sviluppo; BitBake dovrebbe essere utilizzato solo per modifiche veloci.

*Dobbiamo tenere a mente che non tutte le macchine supportate nello Yocto Project, vendor e community utilizzano il framework linux-yocto. Ciò significa che il meccanismo del frammento di configurazione non è disponibile per quelle macchine e dobbiamo fornire un completo file defconfig per personalizzare il loro kernel.*

È possibile trovare la documentazione completa di `linux-yocto`, che copre tutti gli aspetti del framework `linux-yocto` e concetti avanzati di manutenzione del kernel Linux.

Possiamo usare `recipetool` come aiuto per la procedura di creazione di `.bbappend` utilizzando una riga di comando come `$ recipetool newappend -w -e ../meta-newlayer bc` per creare un file `.bbappend` al di sopra della versione corrente di `bc`.

## Sommario

In questo capitolo abbiamo imparato come personalizzare le ricette esistenti utilizzando i file `.bbappend` e trarne vantaggio evitando la duplicazione del codice sorgente. Abbiamo visto come abilitare o disabilitare una funzionalità, come applicare una patch e come modificare la configurazione di BusyBox e del framework `linux-yocto`.

Nel prossimo capitolo, discuteremo di come lo Yocto Project può aiutarci con alcuni aspetti legali della produzione di un sistema basato su Linux utilizzando pacchetti con licenze diverse. Capiremo di quali artefatti abbiamo bisogno e come Poky può essere configurato per generare gli artefatti che dovrebbero essere condivisi come parte del processo di realizzazione della conformità del copyleft.



# 13. La conformità GPL

In questo capitolo vedremo come possiamo garantire la conformità delle licenze open source e come possiamo usare Poky per fornire gli artefatti necessari, come il codice sorgente, il testo della licenza e l'elenco del lavoro derivato. Questo è fondamentale per la maggior parte dei prodotti che vengono introdotti sul mercato al giorno d'oggi, poiché il codice open source deve convivere fianco a fianco con quello proprietario.

## ***Il copyleft***

**Copyleft** è un modo legale di utilizzare la legge sul copyright al fine di massimizzare i diritti ed esprimere la libertà. Ha un impatto così grande sul nostro lavoro quotidiano che le aziende devono sapere come gestire le licenze open source e del software libero, poiché hanno un grande impatto sui loro prodotti.

Quando si crea una distribuzione Linux, vengono utilizzati almeno due progetti: il kernel Linux e un compilatore. Il compilatore più comunemente usato oggi è lo `GNU Compiler Collection (GCC)`.

Il kernel Linux è rilasciato con licenza `GPLv2` e `GCC` è rilasciato con le licenze `GPLv2`, `GPLv2.1` e `GPLv3`, a seconda del progetto utilizzato.

Tuttavia, un sistema basato su Linux può includere virtualmente tutti i progetti disponibili al mondo, oltre a tutte le applicazioni realizzate dall'azienda per il suo prodotto. Come facciamo a sapere il numero di progetti e licenze inclusi e come soddisfiamo i requisiti di conformità al copyleft?

Questo capitolo descrive come lo Yocto Project è d'aiuto nell'attività, ma si tenga presente che si deve sapere esattamente cosa fornire e le possibili incompatibilità di licenza. In caso di dubbi, consultare il proprio ufficio legale o un avvocato specializzato in diritti d'autore.

In questo capitolo, esamineremo come lo Yocto Project può aiutarci con le attività più comuni richieste per la conformità con il copyleft.

## **Conformità al copyleft rispetto al codice proprietario**

È importante comprendere che il codice proprietario e il codice coperto da copyleft possono coesistere nello stesso prodotto. Dobbiamo fare attenzione alle librerie a cui linkiamo il codice perché alcune potrebbero avere problemi di compatibilità delle licenze. Tuttavia, questo è lo standard nella maggior parte dei prodotti disponibili oggi sul mercato.

## **Alcune linee guida per la conformità delle licenze**

Come già accennato, un sistema basato su Linux è un insieme di diversi progetti, ciascuno con una licenza diversa. Lo Yocto Project aiuta gli sviluppatori a capire che la maggior parte degli obblighi di progetto copyleft hanno le seguenti condizioni:

- Il codice sorgente del progetto deve essere fornito insieme al binario
- La licenza del progetto deve essere fornita insieme al binario
- Qualsiasi modifica al progetto o qualsiasi script necessario per configurarlo e compilarlo deve essere fornito insieme al file binario

Ciò significa che se un progetto sotto copyleft viene modificato, il testo della licenza, il codice sorgente di base e qualsiasi modifica devono essere inclusi nel distribuito finale.

Le ipotesi coprono la maggior parte dei diritti garantiti dalle licenze copyleft. Queste sono le parti in cui lo Yocto Project può aiutarci. Tuttavia, prima di rilasciare qualsiasi cosa, si consiglia di controllare tutti i materiali per assicurarsi che siano completi.

## Gestione delle licenze software con Poky

Una caratteristica importante di Poky è la possibilità di gestire le licenze. La maggior parte delle volte, noi sviluppatori non ci preoccupiamo delle licenze perché ci concentriamo sui nostri stessi bug. Tuttavia, durante la creazione di un prodotto, è molto importante prestare attenzione e conoscere le licenze e i tipi di licenze presenti nel prodotto.

Poky tiene traccia delle licenze, lavora con licenze commerciali e non commerciali e ha una strategia per lavorare con applicazioni proprietarie, almeno durante il ciclo di sviluppo.

Una cosa importante da sapere, all'inizio, è che una ricetta viene rilasciata con una certa licenza e rappresenta un progetto rilasciato con una licenza diversa. La ricetta e il progetto sono due entità diverse e hanno licenze diverse, quindi le due diverse licenze devono essere considerate parte del prodotto.

Nella maggior parte delle ricette, l'informazione è un commento contenente il copyright, la licenza e il nome dell'autore; queste informazioni riguardano la ricetta stessa. Poi, c'è un insieme di variabili per descrivere la licenza del pacchetto, e sono le seguenti:

- **LICENSE**: Descrive la licenza con cui è stato rilasciato il pacchetto.
- **LIC\_FILES\_CHKSUM**: Questo potrebbe non sembrare molto utile a prima vista. Descrive il file di licenza e il relativo checksum per un determinato pacchetto e potremmo trovare molte variazioni nel modo in cui un progetto descrive la sua licenza. I file di licenza più comuni sono archiviati in `meta/files/common-licenses/`.

Alcuni progetti includono un file, come `COPYING` o `LICENSE`, che specifica la licenza per il codice sorgente. Altri usano una nota di header in ogni file o nel file `main`. La variabile **LIC\_FILES\_CHKSUM** ha il checksum per il testo della licenza di un progetto; se vengono modificate delle lettere, viene modificato anche il checksum. Questo viene utilizzato per assicurarsi che qualsiasi modifica venga annotata e accettata consapevolmente dallo sviluppatore. Una modifica della licenza può essere una correzione di errore di battitura; tuttavia, potrebbe anche trattarsi di una modifica degli obblighi legali, quindi è importante che lo sviluppatore riveda e comprenda la modifica.

Quando viene rilevato un checksum di licenza diverso, BitBake genera un errore di compilazione e punta al progetto a cui è stata modificata la licenza. È necessario prestare attenzione quando ciò accade poiché la modifica della licenza potrebbe influire sull'uso di questo software. Per poter creare nuovamente qualsiasi cosa, si deve modificare il valore **LIC\_FILE\_CHKSUM** di conseguenza e aggiornare il campo **LICENSE** in modo che corrisponda alla modifica della licenza. Se i termini della licenza sono cambiati, è necessario consultare l'ufficio legale.

### Licenze commerciali

Per default, Poky non installa alcun pacchetto con una restrizione di licenza commerciale. L'esempio più comunemente usato è il pacchetto `gststreamer1.0-plugins-ugly`. Ciò si ottiene tramite una variabile utilizzata da queste ricette con qualche restrizione della licenza; la variabile **LICENSE\_FLAGS** viene utilizzata per determinare la restrizione.

Nel caso di `gststreamer1.0-plugins-ugly`, la variabile nella ricetta è impostata su **LICENSE\_FLAGS = "commercial"**, sebbene possa avere una stringa. Alcuni progetti scelgono di impostarlo su **LICENSE\_FLAGS = "<license>\_\${PN}\_\${PV}"**.

Per installare queste ricette, dobbiamo inserire una "whitelist" di licenze speciali desiderate in `build/conf/local.conf` e possiamo farlo usando **LICENSE\_FLAGS\_WHITELIST**. Questa variabile determina la licenza speciale che può essere utilizzata e ha un contenuto molto flessibile.

Ad esempio, per i plug-in GStreamer Ugly, potremmo volere solo che questo pacchetto sia installato, quindi aggiungiamo la seguente variabile in `build/conf/local.conf`:

```
LICENSE_FLAGS_WHITELIST_pn-gstreamer1.0-plugins-ugly = "commercial"
```

Ciò consente l'uso di `gstreamer1.0-plugins-ugly` escludendo qualsiasi altra ricetta commerciale, come `gstreamer1.0-plugins-bad`. Tuttavia, se vogliamo che BitBake installi qualsiasi pacchetto commerciale dalla nostra immagine, possiamo utilizzare il seguente codice in `build/conf/local.conf`:

```
LICENSE_FLAGS_WHITELIST = "commercial"
```

## Utilizzo di Poky per ottenere la conformità con il copyleft

A questo punto, sappiamo come utilizzare Poky e capiamo il suo obiettivo principale. È tempo di comprendere gli aspetti legali della produzione di un sistema basato su Linux che utilizza pacchetti con licenze diverse.

Possiamo configurare Poky per generare gli artefatti che dovrebbero essere condivisi come parte del processo di conformità del copyleft.

### Controllo delle licenze

Per aiutarci a ottenere la conformità al copyleft, Poky genera un manifest di licenza durante la creazione dell'immagine, che si trova in

```
build/tmp/depoy/licenses/<image_name-machine_name-datestamp>/.
```

Per dimostrare questo processo, utilizzeremo l'immagine `core-image-full-cmdline` per la macchina `qemuarm`. Per iniziare con il nostro esempio, guardare i file in

```
build/tmp/depoy/licenses/core-image-full-cmdline-qemuarm-<datastamp>
```

, che sono i seguenti:

- `image_license.manifest`: Elenca i nomi delle ricette, le versioni, le licenze e i file dei pacchetti che sono disponibili in `build/tmp/depoy/image/<machine>` ma non installati all'interno di `rootfs`. Gli esempi più comuni sono il bootloader, l'immagine del kernel Linux e i file DTB.
- `package.manifest`: Elenca tutti i pacchetti nell'immagine.
- `license.manifest`: Elenca i nomi, le versioni, i nomi delle ricette e le licenze per tutti i pacchetti installati. Questo file può essere utilizzato per il controllo della conformità del copyleft.

### Fornire il codice sorgente

Il modo più ovvio in cui Poky può aiutarci a fornire il codice sorgente di ogni progetto utilizzato nella nostra immagine è condividere il contenuto di `DL_DIR`. Tuttavia, questo approccio ha una trappola importante: qualsiasi codice sorgente proprietario sarà condiviso all'interno di `DL_DIR` se è condiviso così com'è. Inoltre, questo approccio condividerà qualsiasi codice sorgente, comprese le parti che non sono richieste dalla conformità del copyleft.

Un altro modo è configurare Poky per generare il set di codici sorgente e decidere cosa verrà distribuito. Questo può essere fatto usando la classe `archiver`. Questa classe copia il codice sorgente per ogni pacchetto nella cartella `build/tmp/depoy`, separato dall'architettura (nel nostro esempio, le architetture attuali sono `allarch-poky-linux`, `arm-poky-linux-gnueabi` e `x86_64-linux`) e la licenza. Il pacchetto per `armpoky-linux-gnueabi`, rilasciato sotto GPLv3, si trova nella directory `build/tmp/depoy/sources/arm-poky-linux-gnueabi/GPLv3/package-name`.

Poky deve essere configurato per archiviare il codice sorgente prima che venga creata l'immagine finale. Quindi, per averlo, possiamo incollare le seguenti variabili in `build/conf/local.conf`:

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

Si tenga presente che, anche con questo approccio, se condividiamo le directory **build/tmp/deplo**y/**sources**, i sorgenti proprietari vengono condivisi, anche se ciò non è necessario, sebbene ora possiamo scegliere di condividere il sorgente in base alla licenza. Possiamo copiare solo i pacchetti sotto GPLv3 o MIT in un luogo condivisibile, o in qualsiasi altra combinazione di licenze, secondo la strategia di condivisione desiderata.

Un esempio di una riga di comando di copia, usata per copiare tutti i pacchetti sotto qualsiasi licenza GPL (da: <http://www.yoctoproject.org/docs/2.4/dev-manual/dev-manual.html>), è il seguente:

```
# Script to archive a subset of packages matching specific license(s)
# Source and license files are copied into sub folders of package folder
# Must be run from build folder
#!/bin/bash
src_release_dir="source-release"
mkdir -p $src_release_dir
for a in tmp/deplo
```

y/**sources**/\*; do
 for d in \$a/\*; do
 Get package name from path
 p=`basename \$d`
 p=\${p%-\*}
 p=\${p%-\*}
 # Only archive GPL packages (update \*GPL\* regex for your license check)
 numfiles=`ls tmp/deploy/**licenses**/\$p/\*GPL\* 2> /dev/null | wc -l`
 if [ \$numfiles -gt 1 ]; then
 echo Archiving \$p
 mkdir -p \$src\_release\_dir/\$p/source
 cp \$d/\* \$src\_release\_dir/\$p/source 2> /dev/null
 mkdir -p \$src\_release\_dir/\$p/license
 cp tmp/deploy/**licenses**/\$p/\* \$src\_release\_dir/\$p/license 2> /dev/null
 fi
 done
done

Se si preferisce ricevere aiuto da Poky su quale licenza deve avere la nostra attenzione, si può aggiungere il codice **ARCHIVER\_MODE[filter] ?= "yes"** a **build/conf/local.conf**. La configurazione di default prevede di avere il codice sorgente per ogni progetto, in altre parole nessun filtro. Tuttavia, se si preferisce avere solo il codice sorgente dei progetti **COPYLEFT\_LICENSE\_INCLUDE**, si può usare un filtro.

La variabile **COPYLEFT\_LICENSE\_INCLUDE** include attualmente tutte le licenze che iniziano con GPL o LGPL. Questa variabile può essere overridden [sovra-scritta] in **build/conf/local.conf** se desideriamo assicurarci di includere un'altra licenza o variazione.

## Fornitura di script di compilazione e modifiche al codice sorgente

Con la configurazione fornita nella sezione precedente, Poky impacchetta il codice sorgente originale per ogni progetto. Nel caso in cui vogliamo includere il codice sorgente "patchato", useremo solo **ARCHIVER\_MODE[src] = "patched"**; in questo modo, Poky impacchetterà il codice sorgente del progetto dopo il task **do\_patch**. Include modifiche dalle ricette o dal file **bbappend**.

In questo modo, il codice sorgente e qualsiasi modifica possono essere condivisi facilmente. Tuttavia, c'è ancora un tipo di informazioni che non è stato creato finora: la procedura utilizzata per configurare e buildare il progetto.

Per avere un ambiente di build riproducibile, possiamo condividere il progetto configurato, in altre parole, il progetto dopo il task **do\_configure**. Per questo, possiamo aggiungere quanto segue a **build/conf/local.conf**:

```
ARCHIVER_MODE[src] = "configured"
```

È importante ricordare che dobbiamo considerare che la persona dall'altra parte potrebbe non utilizzare lo Yocto Project per la conformità al copyleft; in alternativa, se lo stanno utilizzando, devono sapere che la modifica apportata al codice sorgente originale e alla

procedura di configurazione non è disponibile. Questo è il motivo per cui condividiamo il progetto configurato: consente a chiunque di riprodurre il nostro ambiente di build.

Per tutte le versioni del codice sorgente, il file risultante predefinito è un tarball; altre opzioni aggiungeranno `ARCHIVER_MODE[srpm] = "1"` a `build/conf/local.conf` e il file risultante sarà un pacchetto `SRPM`.

### Fornire il testo della licenza

Quando si fornisce il codice sorgente, il testo della licenza viene condiviso al suo interno. Se vogliamo il testo della licenza all'interno della nostra immagine finale, possiamo aggiungere quanto segue a `build/conf/local.conf`:

```
COPY_LIC_MANIFEST = "1"
COPY_LIC_DIRS = "1"
```

In questo modo, i file di licenza verranno inseriti nel filesystem di root, in `/usr/share/common-licenses/`.

## Sommario

In questo capitolo, abbiamo appreso come Poky può aiutare con la conformità della licenza copyleft e abbiamo anche appreso perché non dovrebbe essere utilizzato come base legale. Poky ci consente di generare codice sorgente, script di riproduzione e testo di licenza per i pacchetti utilizzati nella distribuzione. Inoltre, abbiamo appreso che il manifesto della licenza generato all'interno dell'immagine può essere utilizzato per controllare l'immagine.

Nel prossimo capitolo impareremo come utilizzare gli strumenti dello Yocto Project con hardware reale. Utilizzeremo lo Yocto Project per generare un'immagine da utilizzare con le macchine Beagle Bone Black, Raspberry Pi e Wandboard.

# 14. Avvio di Linux Embedded Custom

È ora! Ora siamo pronti per avviare il nostro Linux embedded personalizzato, poiché abbiamo appreso i concetti richiesti e acquisito conoscenze sufficienti sullo Yocto Project e su Poky. In questo capitolo, metteremo in pratica ciò che abbiamo imparato finora sull'uso di Poky con layer BSP esterni, lo useremo per generare un'immagine da utilizzare con macchine BeagleBone Black, Raspberry Pi 3 e Wandboard e lo avvieremo utilizzando la scheda SD.

Gli stessi concetti possono essere applicati a tutte le altre schede, a condizione che il fornitore fornisca un livello BSP da utilizzare con lo Yocto Project.

Vedremo un elenco dei layer BSP più comunemente usati in questo capitolo. Questo non dovrebbe essere considerato come un elenco completo o definitivo, ma vogliamo facilitare la ricerca del layer necessario nel caso in cui si abbia una scheda di un fornitore specifico. Questo elenco è il seguente, in ordine alfabetico:

- `Allwinner`: Questo ha il layer meta-allwinner
- `BeagleBoard`: Questo ha il layer meta-ti
- `CuBox-i`: Questo ha il layer meta-freescale-3rdparty
- `Intel`: Questo ha il layer meta-intel
- `Raspberry Pi`: Questo ha il layer meta-raspberrypi
- `Texas Instruments`: Questo ha il layer meta-ti
- `Wandboard`: Questo ha il layer meta-freescale-3rdparty

## ***Esplorazione delle schede***

Per facilitare l'esplorazione delle capacità dello Yocto Project, è bene avere una scheda reale in modo da poter godere dell'esperienza di avvio del nostro sistema embedded personalizzato. Per questo, abbiamo cercato di raccogliere le schede più comunemente utilizzate e ampiamente disponibili in modo che le possibilità che se ne possieda una siano maggiori.

Questo capitolo tratterà i passaggi per le seguenti schede:

- `BeagleBone Black`: BeagleBone Black è basato sulla comunità, con membri in tutto il mondo. Ulteriori informazioni sono disponibili su <https://beagleboard.org/black/>.
- `Raspberry Pi 3`: La più famosa scheda basata su ARM, con la più ampia community riunita in tutto il mondo. Maggiori informazioni su <https://www.raspberrypi.org/>.
- `Wandboard`: Wandboard è supportato dalla community di Wandboard. Maggiori informazioni sono disponibili su <http://www.wandboard.org/>.

Tutte le schede elencate sono gestite da organizzazioni senza fini di lucro basate sull'istruzione e sul tutoraggio, il che rende la comunità un luogo fertile per scoprire il mondo di Linux embedded. La figura seguente riassume le schede e le loro caratteristiche principali:

Versione Scheda	Funzionalità
BeagleBone Black	TI AM335x (single-core) 512MB DDR3 RAM
Raspberry Pi 3	Broadcom BCM2837 64bit CPU (quad-core) 1 GB RAM 802.11ac wireless and Bluetooth
Wandboard Solo	NXP i.MX6S processor (single-core) 512 MB RAM
Wandboard Dual	NXP i.MX6DL processor (dual-core) 1 GB RAM 802.11ac wireless and Bluetooth
Wandboard Quad	NXP i.MX6Q processor (quad-core) 2 GB RAM 802.11ac wireless and Bluetooth SATA
Wandboard QuadPlus	NXP i.MX6QP processor (quad-core) 2 GB RAM 802.11ac wireless and Bluetooth SATA

## Alla scoperta del livello BSP giusto

Nel [Capitolo 10](#), *Esplorazione di Layer Esterni*, abbiamo appreso che lo Yocto Project consente la suddivisione dei suoi metadati tra layer diversi. Organizza i metadati in modo che possiamo scegliere quale meta layer esatto aggiungere al progetto.

Il modo per trovare il BSP per una scheda varia, ma generalmente possiamo trovarlo visitando <http://layers.openembedded.org/>. Possiamo cercare il nome della macchina e il sito Web trova nel suo database quale layer lo contiene.

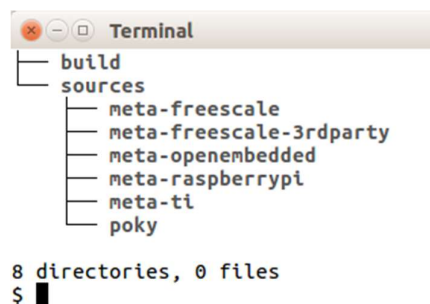
Nelle prossime sezioni descriveremo i passaggi necessari per passare dal codice sorgente al binario finale da copiare sulla scheda. Si può liberamente saltare una sezione se la scheda non è disponibile.

## Il Baking per l'hardware

Dopo aver scoperto il layer BSP per l'hardware che utilizzeremo per la build, è necessario scaricare tutti i meta layer necessari e creare l'ambiente di build.

Prima di iniziare, dobbiamo assicurarci che tutti i requisiti di sistema siano soddisfatti. Abbiamo discusso di questi requisiti nel [Capitolo 2](#), *Creazione del Sistema Poky-Based*.

L'uso dei meta layer ci costringe a gestire molti repository Git di metadati. Un buon modo per evitare confusione è mettere tutte i sorgenti relativi a quei meta layer in una directory specifica. La figura seguente ne mostra un esempio:



È consigliabile mantenere aggiornati i layer all'interno dei sorgenti, poiché si apportano correzioni di sicurezza, correzioni di bug e nuove funzionalità.

Esistono diversi modi per gestire i layer multipli durante la creazione di un prodotto. Un'opzione è utilizzare il combo-layer (<https://wiki.yoctoproject.org/wiki/Combo-layer>), che replica i commit in un singolo albero Git, così come “git submodules” e “repo” per gestire più repository Git in un modo più conveniente. Quando si utilizza repo, ci sono facilitazioni per i progetti “long-term”, ad esempio la creazione di un file manifest che elenca tutti i meta layer da clonare, vedere <http://doc.ossystems.com.br/managing-platforms.html> su come creare la struttura repo. Quale approccio utilizzare per organizzare i metadati è una preferenza personale e la scelta di avere tutti i repository Git dei meta layer clonati solo manualmente è buona per casi d'uso semplici.

Nelle prossime sezioni creeremo la struttura per le tre schede a cui si fa riferimento in questo libro.

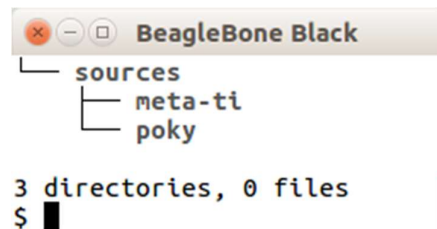
## Baking per BeagleBone Black

Per aggiungere questo supporto per la scheda al progetto, dobbiamo includere il meta layer `meta-ti`, che è il layer BSP con supporto per diverse schede TI, inclusa la BeagleBone Black, ma non solo. È possibile accedere al meta layer all'indirizzo <http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti>.

Per creare la struttura dei sorgenti, eseguire le seguenti righe di comando:

```
$ mkdir sources
$ cd sources
$ git clone --branch rocko git://git.yoctoproject.org/poky
$ git clone --branch rocko git://git.yoctoproject.org/meta-ti
```

La struttura di directory finale è mostrata nella figura seguente:



Dopo aver completato questo, dobbiamo creare la directory `build` e aggiungere il livello BSP. Possiamo farlo usando le seguenti righe di comando:

```
$ cd ..
$ source sources/poky/oe-init-build-env build
$ bitbake-layers add-layer ../sources/meta-ti
```

Dopo aver impostato correttamente la directory `build` e i layer BSP, possiamo avviare la build. All'interno della directory `build`, dobbiamo chiamare il seguente comando:

```
$ MACHINE=beaglebone bitbake <image>
```

La variabile `MACHINE` può essere modificata a seconda della scheda che vogliamo usare o impostata in `build/conf/local.conf`.

Se vogliamo il building di `core-image-sato`, che fornisce un ambiente grafico embedded, dovremmo eseguire il seguente comando:

```
$ MACHINE=beaglebone bitbake core-image-sato
```

## Baking per Raspberry Pi 3

Per aggiungere il supporto di questa scheda al nostro progetto, dobbiamo includere il layer `meta-raspberrypi`, che è il layer BSP con supporto per le schede Raspberry Pi, incluso il Raspberry Pi 3, ma non limitato a questo. Il meta layer è disponibile all'indirizzo <http://git.yoctoproject.org/cgit/cgit.cgi/meta-raspberrypi>.

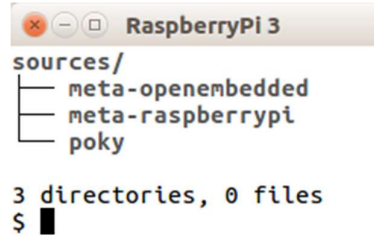
Per creare la struttura dei sorgenti, eseguire le seguenti righe di comando:

```
$ mkdir sources
$ cd sources
```



```
$ git clone --branch rocko git://git.yoctoproject.org/poky
$ git clone --branch rocko git://git.yoctoproject.org/meta-raspberrypi
$ git clone --branch rocko git://git.openembedded.org/meta-openembedded
```

La struttura di directory finale è mostrata nella figura seguente:



Dopo aver completato questo, dobbiamo creare la directory `build` e aggiungere il livello BSP. Possiamo farlo usando le seguenti righe di comando:

```
$ cd ..
$ source sources/poky/oe-init-build-env build
$ bitbake-layers add-layer ../sources/meta-openembedded/meta-oe
$ bitbake-layers add-layer ../sources/meta-openembedded/meta-python
$ bitbake-layers add-layer ../sources/meta-raspberrypi
```

Con la directory `build` e i layer BSP impostati correttamente, possiamo avviare la build. All'interno della directory `build`, dobbiamo chiamare il seguente comando:

```
$ MACHINE=raspberrypi3 bitbake <image>
```

La variabile `MACHINE` può essere modificata a seconda della scheda che vogliamo usare o impostata in `build/conf/local.conf`.

Se vogliamo il building di `core-image-sato`, che fornisce un ambiente grafico embedded, dovremmo eseguire il seguente comando:

```
$ MACHINE=raspberrypi3 bitbake core-image-sato
```

## Baking per la Wandboard

Per aggiungere il supporto per la scheda al progetto, dobbiamo includere il meta layer `meta-freescale-3rdparty`, che è il layer BSP con supporto per Wandboard, ma non limitato a questo. È possibile accedere al meta layer su <https://github.com/Freescale/meta-freescale-3rdparty>.

Il `meta-freescale-3rdparty` dipende dal meta layer di `meta-freescale`, quindi dobbiamo aggiungere entrambi al nostro progetto.

Per creare la struttura dei sorgenti, eseguire le seguenti righe di comando:

```
$ mkdir sources
$ cd sources
$ git clone --branch rocko git://git.yoctoproject.org/poky
$ git clone --branch rocko https://github.com/Freescale/meta-freescale-3rdparty.git
$ git clone --branch rocko git://git.yoctoproject.org/meta-freescale.git
```

La struttura di directory finale è mostrata nella figura seguente:



Dopo aver completato questo, dobbiamo creare la directory `build` e aggiungere il livello BSP. Possiamo farlo usando le seguenti righe di comando:

```
$ cd ..
$ source sources/poky/oe-init-build-env build
$ bitbake-layers add-layer ../sources/meta-freescale
$ bitbake-layers add-layer ../sources/meta-freescale-3rdparty
```

Alcuni pacchetti inclusi in NXP ARM BSP hanno sono proprietari e sono seguiti da un `end-user license agreement (EULA)` che mostra l'impatto legale del suo utilizzo.

Principalmente, i driver GPU, i codec VPU/IPU e il layer `meta-freescale` hanno un file EULA che descrive i diritti e gli obblighi per utilizzare i binari e il sorgente. Leggere altro sull'EULA e, nel caso lo si accetti, modificare il file `build/conf/local.conf` per impostare `ACCEPT_FSL_EULA` a 1, come mostrato nella seguente riga di codice:

```
ACCEPT_FSL_EULA = "1"
```

Questo non è necessario per il funzionamento della scheda, ma per un utilizzo completo le caratteristiche hardware sono indispensabili.

Con la directory `build` e i layer BSP impostati correttamente, possiamo avviare la build. All'interno della directory `build`, dobbiamo chiamare il seguente comando:

```
$ MACHINE=wandboard bitbake <image>
```

La variabile `MACHINE` può essere modificata a seconda della scheda che vogliamo usare o impostata in `build/conf/local.conf`.

Se vogliamo il building di `core-image-sato`, che fornisce un ambiente grafico embedded, dovremmo eseguire il seguente comando:

```
$ MACHINE=wandboard bitbake core-image-sato
```

## Boot dell'immagine

Il processo di build richiederà probabilmente del tempo. C'è un'enorme quantità di lavoro svolto dietro le quinte, ma è un processo semplice.

Al termine della build, è necessario distribuire l'immagine generata sulla scheda, questo è un processo che varia da una scheda all'altra. Tratteremo le istruzioni per ciascuna scheda nelle sezioni seguenti.

### Boot di BeagleBone Black dalla scheda SD

Al termine del processo di build, l'immagine sarà disponibile all'interno della directory `build/tmp/deploy/images/beaglebone/`. Ci sono molti file, ma il BSP Texas Instrument genera un'immagine della scheda SD pronta per l'uso.

Il file che vogliamo usare è `core-image-sato-beaglebone.wic`.

Puntare al dispositivo giusto e controllare di non scrivere sul proprio hard disk.

Per copiare l'immagine `core-image-sato` sulla scheda SD, dovremmo usare l'utilità `dd`, in questo modo:

```
$ sudo dd if=core-image-sato-beaglebone.wic of=/dev/sdX bs=1M
```

Dopo aver copiato il contenuto sulla scheda SD, inserirlo nello slot della scheda SD, collegare il cavo HDMI e accendere la macchina. Dovrebbe partire bene.

### Boot di Raspberry Pi 3 dalla scheda SD

Al termine del processo di build, l'immagine sarà disponibile all'interno della directory `build/tmp/deploy/images/raspberrypi3/` directory. Ci sono molti file, ma il bsp del Raspberry Pi genera un'immagine della scheda SD pronta per l'uso.

Il file che vogliamo usare è `core-image-sato-raspberrypi3.rpi-sdimg`.

Puntare al dispositivo giusto e controllare di non scrivere sul proprio hard disk.

Per copiare l'immagine `core-image-sato` sulla scheda SD, dovremmo usare l'utilità `dd`, in questo modo:

```
$ sudo dd if=core-image-sato-raspberrypi3.rpi-sdimg of=/dev/sdX bs=1M
```

Dopo aver copiato il contenuto sulla scheda SD, inserirlo nello slot della scheda SD, collegare il cavo HDMI e accendere la macchina. Dovrebbe partire bene.

## Boot di Wandboard dalla scheda SD

Al termine del processo di build, l'immagine sarà disponibile all'interno della directory `build/tmp/deploy/images/wandboard/`. Ci sono molti file, ma il BSP NXP ARM genera un'immagine della scheda SD pronta per l'uso.

Il file che vogliamo usare è `core-image-sato-wandboard.wic.gz`.

*Puntare al dispositivo giusto e controllare di non scrivere sul proprio hard disk.*

Per copiare l'immagine `core-image-sato` sulla scheda SD, dovremmo usare l'utilità `dd`, in questo modo:

```
$ gunzip core-image-sato-wandboard.wic.gz
$ sudo dd if=core-image-sato-wandboard.wic of=/dev/sdX bs=1M
```

Dopo aver copiato il contenuto sulla scheda SD, inserirlo nello slot della scheda SD, collegare il cavo HDMI e accendere la macchina. Dovrebbe partire bene.

Ci sono due slot per schede SD in una Wandboard. Lo slot principale si trova nella scheda CPU, utilizzato per il boot, e uno slot secondario si trova nella scheda periferica (la scheda di base).

## Passi successivi

Uffa! Ce l'abbiamo fatta! Ora si dovrebbero conoscere le basi del sistema di build dello Yocto Project ed essere in grado di estendere le conoscenze a riguardo per coprire altre aree con molto meno problemi. Abbiamo cercato di coprire le attività più comuni nel lavoro quotidiano utilizzando lo Yocto Project e ci sono alcune cose su cui esercitarsi:

- Creazione di file `bbappend` per applicare patch o apportare altre modifiche a una ricetta
- Creare immagini personalizzate
- Modificare il file di configurazione del kernel Linux (`defconfig`)
- Modificare la configurazione di `busybox` e includere i frammenti di configurazione per aggiungere o rimuovere una funzionalità in un layer
- Aggiungere una nuova ricetta per un pacchetto
- Creare un layer di prodotto con macchine, ricette e immagini specifiche del prodotto

Da ricordare che il codice sorgente è la fonte di conoscenza definitiva, quindi va usata. Quando si cerca come fare qualcosa, trovare una ricetta simile fa guadagnare molto tempo per testare approcci diversi per risolvere il problema.

Alla fine, probabilmente ci si troverà nella posizione di correggere o migliorare qualcosa su OpenEmbedded-Core, un meta layer o in un BSP. Non c'è da aver paura, inviare le patch e prendere i feedback e le richieste di modifiche come un'opportunità per imparare e migliorare il modo di risolvere un problema.

## Sommario

In questo capitolo finale, abbiamo imparato come trovare il BSP per una scheda da utilizzare nel progetto. Abbiamo consolidato la nostra conoscenza dello Yocto Project aggiungendo layer BSP esterni e utilizzandoli in schede reali con un'immagine generata.

In tutto il libro, abbiamo acquisito le informazioni di base necessarie affinché si possa apprendere qualsiasi altro aspetto dello Yocto Project di cui si potrebbe aver bisogno. Si ha una comprensione generale di ciò che accade dietro le quinte quando si chiede a BitBake di creare una ricetta o un'immagine. D'ora in poi, si è pronti per liberare la mente e provare cose nuove. Ci sono un sacco di schede disponibili, che aspettano che gli si dia vita. La palla è nel tuo campo ora; ecco dove inizia il divertimento!