POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Generazione automatica distribuzione Linux per piattaforme Embedded

Relatore:
Prof. Gianpiero Cabodi

Candidato:
Luca Cucci
matricola: 215583

Supervisore aziendale
Magneti Marelli
Ing. Fabio Tosetto

Anno accademico 2016-2017

Ringraziamenti

Sommario

La tematica principale di questo lavoro di tesi si focalizza sulla compilazione software per piattaforme Linux embedded impiegate nel settore automotive. Nella prima parte della tesi si fornisce una panoramica dei sistemi embedded e delle motivazioni che spingono ad adottare distribuzioni Linux per queste piattaforme. In seguito vengono analizzati quattro tool opensource di generazione automatica di cross-compilatori, confrontandoli in base alle funzionalità offerte. Nella fattispecie devono essere in grado di generare toolchain di cross-compilazione e root filesystem, essere integrati in ambienti di sviluppo ed infine fornire un web frontend che permette di cross-compilare da remoto. L'esito di tale analisi individua nel progetto Yocto la scelta più appropriata per lo svolgimento della tesi. Successivamente si descrivono le caratteristiche hardware del target, ossia la board Udoo Quad, e la progettazione della struttura software da integrare all'interno di una distribuzione Linux generata per tale board. In seguito si descrivono i passi per installare, configurare ed utilizzare Yocto. In particolare quest'ultimo è stato utilizzato per: generare una distribuzione Linux per la Udoo Quad ed integrare un framework proprietario di Magneti Marelli S.p.A all'interno della distribuzione stessa. Lo scopo di tali test è quello di mostrare un possibile utilizzo del progetto Yocto, ma soprattutto evidenziare i vantaggi di cross-compilare software tramite il progetto stesso. Il risultato generale ottenuto evidenzia che il tempo totale per generare la distribuzione è di 19 ore e 25 minuti. Il vantaggio di avere un processo così lungo è che Yocto può conservare lo stato di tutti i passi intermedi, offrendo la possibilità di riutilizzarli per future compilazioni, accorciando notevolmente le tempistiche del processo. Ad esempio il tempo di cross-compilazione al fine di integrare parte del framework all'attuale distribuzione sono dell'ordine dei minuti, dunque con un guadagno medio di circa il 95%. Il lavoro si è concluso quindi con la scrittura dell'immagine della distribuzione su SD card, che funge da dispositivo di avvio dalla Udoo Quad. Infine è stato redatto un manuale utente all'interno del quale sono dettagliati i passi svolti per configurare ed utilizzare il progetto, mentre in Appendice sono riportati i file modificati nel processo di configurazione.

Indice

Elenco delle figure

Acronimi

ABS Anti-lock Braking System

API Application Programming Interface

ARIB Association of Radio Industries and Businesses

ATM Automated Teller Machine

BSM Basic Safety Message

BSP Board Support Package

C-V2X Cellular V2X

CAN Controller Area Network

CAM Cooperative Aware Message

DENM Decentralized Environmental Notification Message

DSRC Dedicated Short-Range Communications

D2D Device-to-Device

D2N Device-to-Network

ECU Engine Control Unit

ESC Electronic Stability Control

ETSI European Telecommunications Standards Institute

FHS Filesystem Hierarchy Standard

GUI Graphical User Interface

HMI Human Machine Interface

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

ITS Intelligent Transportation Systems

LDM Local Dynamic Map

MAP MAP Data

MMU Memory Management Unit

OE OpenEmbedded

POTI Positioning Timing

SDK Software Development Kit

SPAT Signal Phase and Time

S&R Sender&Receiver

TIM Traveler Information Message

V2I Vehicle-to-Infrastructure

V2N Vehicle-to-Network

V2P Vehicle-to-Pedestrian

V2V Vehicle-to-Vehicle

V2X Vehicle-to-Everything

VDP Vehicle Data Provider

WAVE Wireless Access in Vehicular Environment

WSA WAVE Service Advertisment

3GPP 3rd Generation Partnership Project

Capitolo 1

Introduzione

I sistemi embedded hanno ormai rivoluzionato il modo del vivere quotidiano delle persone. Essi sono nati come semplici dispositivi specializzati a compiere precise funzioni, dotati di pochi componenti hardware ed un software relativamente semplice. Successivamente sono evoluti sia come componenti hardware che come software. In generale, all'aumentare della complessità delle funzioni da compiere corrisponde un incremento della complessità dell'hardware del sistema embedded da adoperare. Ciò si riflette anche sulla relativa parte software. Molti sistemi embedded necessitano ad esempio di essere equipaggiati con un sistema operativo, tipicamente della famiglia Linux. Esistono strumenti noti con il nome di toolchain di cross-compilazione che sono in grado di generare una versione Linux ottimizzata per sistemi embedded. Esse permettono di compilare software per un determinato target hardware utilizzando una macchina, nota come macchina build, differente dal target stesso. il sistema build potrebbe ad esempio essere un PC della famiglia x86, mentre il target un microprocessore della famiglia ARM.

Oggigiorno la diffusione dei sistemi embedded è capillare in quanto maggior parte dei dispositivi elettronici sono basati su queste piattaforme; di fatti si possono trovare all'interno di smartphone, TV, tablet, ATM ed elettrodomestici. La loro presenza non si limita però a dispositivi ad uso del consumatore, ma coinvolge svariati settori: dal campo medico al campo della produzione industriale fino al settore automobilistico. Con il passare degli anni, nel settore automobilistico, sono stati introdotti nuovi sistemi elettronici ed alcuni sistemi meccanici si sono tramutati in sistemi elettromeccanici. Tipici esempi sono: ABS, ESC, sensori di parcheggio, sistemi di navigazione assistita ed Engine Control Unit (ECU). Queste ultime si occupano della gestione elettronica-digitale per la formazione e combustione della miscela, e del contenimento delle emissioni inquinanti di un motore a combustione

interna. Le ECU più complesse controllano inoltre il tempo d'iniezione, la fasatura di distribuzione e d'accensione, e tutte le periferiche del sistema di controllo.

Negli ultimi anni l'automobile è inoltre diventata una sorta di dispositivo interconnesso. Questo è avvenuto grazie al costante avanzamento tecnologico nel campo delle telecomunicazioni ed alla maggiore integrazione tra i vari dispositivi elettronici dell'automobile ed infrastrutture centrali. In questo modo si crea un sistema di comunicazione intelligente o Intelligent Transportation Systems (ITS)[1] al fine di migliorare la sicurezza stradale, la gestione del traffico e la sensazione di comfort dei conducenti, aiutandoli a prendere la migliore decisione possibile in base alle varie situazioni. I sistemi di comunicazione interveicolare si possono distinguere in [2]: Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I), Vehicle-to-Pedestrian (V2P), ma più in generale si parla di Vehicle-to-Everything (V2X).

Nel corso degli anni, per regolare il mondo delle comunicazioni interveicolari, sono stati emanati diversi standard da varie organizzazioni mondiali. Negli USA è in vigore lo standard Wireless Access in Vehicular Environment (WAVE)[3] emanato da Institute of Electrical and Electronics Engineers (IEEE), in Europa vige lo standard ITS-G5[4] emanato da European Telecommunications Standards Institute (ETSI)[2], mentre in Giappone l'Association of Radio Industries and Businesses (ARIB)[5] ha emanato lo standard STD-T109[6].

Magneti Marelli

La fondazione dell'azienda risale al 1919, col nome di Fabbrica Italiana Magneti Marelli (F.I.M.M.), frutto di una cooperazione tra la FIAT e la Ercole Marelli. Il primo stabilimento era localizzato a Sesto San Giovanni, alle porte di Milano. Magneti Marelli[7] opera a livello internazionale come fornitore di prodotti soluzioni e sistemi ad alta tecnologia per il mondo automotive. La sede centrale è in Italia, a Corbetta, Milano. Con 7,9 miliardi di Euro di fatturato nel 2016, circa 43.000 addetti, 86 unità produttive e 14 centri ricerca e sviluppo, il gruppo è presente in modo capillare in 5 continenti e in 19 Paesi. Magneti Marelli rifornisce i maggiori car makers in Europa, Nord e Sud America e Asia. Nella sua missione di componentista automotive a livello globale, mira a coniugare qualità e offerta competitiva, tecnologia e flessibilità, con l'obiettivo di rendere disponibili prodotti d'eccellenza a costi competitivi. Magneti Marelli punta a valorizzare, attraverso un processo di innovazione continua, conoscenze e competenze trasversali al fine di sviluppare sistemi e soluzioni che contribuiscano

all'evoluzione della mobilità secondo criteri di sostenibilità ambientale, sicurezza e qualità della vita all'interno dei veicoli. Magneti Marelli fa parte del gruppo FCA ed opera a livello internazionale attraverso otto aree di business:

- Electronic Systems: quadri di bordo, infotainment e telematica, lighting e body electronics
- Automotive Lighting: sistemi di illuminazione
- Powertrain: sistemi controllo motore benzina, diesel e multi-carburante; cambio robotizzato AMT Freechoice
- Suspension Systems: sistemi sospensioni, ammortizzatori, dynamic system, ovvero sistemi di controllo dinamico del veicolo
- Exhaust systems: sistemi di scarico, convertitori catalitici, silenziatori
- Motorsport: sistemi elettronici ed elettromeccanici specifici per le competizioni con leadership tecnologica in Formula1, MotoGP, SBK e WRC
- Plastic Components and Modules: componenti e moduli plastici per l'automotive
- Aftermarket Parts and Services: distribuzione ricambi per l'Independent Aftermarket (IAM); Rete Assistenza e Officine Checkstar.

Magneti Marelli è una delle compagnie che sta lavorando ad uniformare gli standard sulla comunicazione interveicolare tramite lo sviluppo di un apposito Framework.

Obiettivo

L'obiettivo principale della tesi è stato quello di generare una piattaforma Linux embedded in grado di supportare il Framework Magneti Marelli tramite sistema di build automatico. In particolare il sistema in esame deve soddisfare determinati requisiti:

- deve essere facilmente utilizzabile anche da utenti non esperti, nascondendo loro una serie di dettagli tecnici configurati a priori, infatti a fronte di un ambiente opportunamente configurato anche un utente non esperto può generare facilmente la propria distribuzione. Questa caratteristica permette di evitare la cross-compilazione manuale del software
- deve fornire un insieme di tool che devono cooperare tra loro aumentando le funzionalità della toolchain stessa. Basandosi sullo stesso back-end deve fornire almeno due interfacce che permettono di generare distribuzioni: una

accessibile da terminale ed una tramite interfaccia web. Quest'ultima deve garantire anche la possibilità di gestire il processo di generazione da remoto e fornire una serie di dettagli ed informazioni che permettono di monitorare lo stato delle operazioni svolte

- deve fornire dei plug-in per ambienti di sviluppo integrato, noti anche con il termine inglese di Integrated Development Environment (IDE), che permettono di utilizzare la toolchain in un ambiente che semplifica la programmazione, ad esempio tramite autocompletamento del codice.

Durante la prima parte della tesi sono stati confrontati quattro generatori di toolchain di cross-compilazione, al fine di individuare la scelta migliore per l'obiettivo preposto.

In seguito è stato scelto il target hardware utilizzato per effettuare i test tramite il tool Yocto, ovvero la board commerciale Udoo Quad Core[8]. La scelta di questa board è legata alla compatibilità hardware con una board proprietaria Magneti Marelli.

Il passo successivo è stato progettare il layer per il Framework Magneti Marelli e selezionare il layer Board Support Package (BSP) per la Udoo Quad; entrambi sono stati integrati all'interno della distribuzione da creare con Yocto. Il layer inerente il Framework deve avere una struttura ben determinata in quanto in prima analisi tiene conto della tecnologia di comunicazione utilizzata, ovvero: V2X, C-V2X[9] e 5G. In secondo luogo le struttura è suddivisa considerando gli standard che regolano le comunicazioni interveicolari, distinguendo tra: standard americano (WAVE), europeo (ETSI) e giapponese (ARIB). Un BSP è invece una collezione di informazioni che definiscono come supportare una particolare piattaforma hardware; include informazioni riguardanti: caratteristiche dell'hardware presente nella piattaforma, il bootloader, il kernel ed i device driver. Ogni piattaforma ha un suo BSP.

Avendo a disposizione tutte le informazioni necessarie, lo step successivo è stato installare e configurare il tool nel sistema utilizzato per la cross-compilazione. Sono stati inoltre configurati il plug-in per integrare Yocto in Eclipse ed il web front-end, ovvero Toaster.

Il lavoro si è concluso con la generazione di una distribuzione Linux embedded effettuata tramite Yocto per la Udoo Quad. All'interno della distribuzione sono stati integrati ed appositamente cross-compilati i moduli del Framework. L'obiettivo di questa prova è stato quello di mostrare le funzionalità ed i vantaggi di Yocto ponendo particolare attenzione alle prestazioni del processo di build. L'immagine della distribuzione è stata infine scritta su una SD card, usata come dispositivo di boot dalla board.

Capitolo 2

Linux nei sistemi embedded

In questo capitolo si affronta una panoramica relativa ai sistemi embedded Linux based. Inizialmente si definiscono i vantaggi relativi all'adozione di Linux per queste piattaforme. Successivamente si effettua una panoramica generale sia delle caratteristiche hardware che dell'architettura software inerente di un sistema di sviluppo Linux embedded.

2.1 Vantaggi Linux

Il termine *embedded Linux* viene adottato per definire i sistemi embedded basati sul kernel Linux ed altre componenti open source. Nel corso degli anni Linux si è affermato come migliore sistema operativo da installare su dispositivi embedded. La motivazione principale è dovuta al fatto che Linux è un sistema operativo open source e completamente free. Un software si definisce free quando la licenza sotto la quale è rilasciato permette di: eseguire il software per qualunque scopo, studiarlo e apportare modifiche, e ridistribuirlo anche quando viene modificato. Tutto ciò comporta i seguenti vantaggi:

- componenti ri-utilizzabili: essendo Linux open source è molto probabile che componenti standard siano già stati sviluppati. Questa caratteristica permette di sviluppare prodotti sempre più complessi basandosi su componenti esistenti
- basso costo: il software, essendo free, può essere duplicato su più dispositivi, riducendo i costi di sviluppo

- maggiore controllo: essendo open source, si possiede il codice sorgente di tutti i componenti del sistema, quindi si ha piena conoscenza delle operazioni svolte dal codice
- qualità: molti componenti open source sono usati in milioni di sistemi in commercio, quindi la qualità deve essere garantita
- fase di test delle nuove features facilitato: la natura free del software permette agli utenti di studiare i nuovi componenti quando vengono rilasciati, permettendo di risolvere velocemente eventuali problemi
- supporto della community: i vari componenti software sono sviluppati da una comunità di sviluppatori che possono fornire supporto di alta qualità.

2.2 Hardware dei sistemi Linux Embedded

Un sistema Linux embedded è costituito dai seguenti componenti: processore, memoria centrale (RAM), memoria di massa (tipicamente flash), bus di comunicazione (I2C, SPI, CAN, 1-wire, SDIO, USB), periferici di comunicazione (Ethernet, Wi-fi, bluetooth).

2.2.1 Processori

I processori supportati dal kernel Linux possono essere divisi in tre categorie:

- processori stand-alone: sono la categoria più performante in quanto sono processori dedicati esclusivamente alla funzione di calcolo. Questo significa che necessitano di circuiteria esterna in modo da integrare determinate funzioni; tipici esempi sono: controller DRAM, sistema di bus e periferiche esterne come tastiera e porte seriali. Nei sistemi embedded possono essere usati anche processori da 32 e 64 bit. Esempi di processori stand-alone sono appartenenti alle famiglie Intel, IBM e Freescale
- processori integrati o System-on-Chip (SoC): sono la categoria maggiormente aottata nei sistemi embedded. All'interno di una board, oltre al processore, sono contenuti anche tutti gli elementi di contorno: il chipset (componente che si occupa di smistare e dirigere il traffico di informazioni all'interno del SoC), i dispositivi di I/O e i vari controller (ad esempio il controller della memoria RAM). Esempi di SoC sono le architetture PowerPC, ARM e Freescale

- altre architetture: oltre alle due grandi famiglie sopra citate, il kernel Linux supporta altre famiglie di processori. Alcuni esempi sono le piattaforme cPCI e ATCA.

2.2.2 Memoria centrale

Il tipo e dimensione della memoria centrale di una piattaforma embedded varia in base alla tipologia del sistema ed alle applicazioni da eseguire. Un sistema Linux embedded molto semplice potrebbe lavorare anche con 8 MB di memoria RAM, ma i sistemi più comuni sono forniti di almeno 32 MB. Ad esempio all'interno di smartphone e tablet le memorie sono ormai dell'ordine dei GB. Inoltre se il sistema è un real-time si potrebbero utilizzare tipologie di memoria più performanti (visti i vincoli di tempo da rispettare), altrimenti si potrebbero usare memorie meno performanti risparmiando così sul costo della board.

2.2.3 Storage

Tipicamente i sistemi embedded possiedono risorse di storage molto limitate. Le memorie flash sono una tecnologia molto adottata ed offrono i seguenti vantaggi: non contengono parti meccaniche, sono abbastanza robuste ed operano ad un determinato voltaggio, garantendo un certo risparmio in termini di consumo energetico. Questo ultimo punto è cruciale nelle piattaforme embedded in quanto sono sistemi generalmente alimentati a batteria.

2.2.4 Gesttione memoria

La memoria viene gestita dal sistema operativo come se fosse un singolo spazio di indirizzamento virtuale la cui dimensione dipende dagli indirizzi del processore. Ad esempio un sistema a 32 bit può indirizzare al massimo 4 GB di memoria. Lo spazio di indirizzamento è suddiviso in varie sezioni. RAM, flash ed indirizzi dei periferici hanno una porzione di memoria dedicata. In questo contesto un dispositivo hardware molto importante è la Memory Management Unit (MMU), che si occupa della traduzione da indirizzi fisici a logici. Essa offre al kernel i seguenti vantaggi:

- possibilità di poter indirizzare una quantità di memoria superiore a quella fisica. In questo caso è necessario avere anche un meccanismo di segmentazione e paginazione della memoria. Tipicamente nei sistemi embedded i sistemi di segmentazione e paginazione sono disabilitati in quanto hanno risorse limitate

- netta separazione tra i vari spazi di indirizzamento allocati ai processi e quello allocato al kernel. Fallimenti in un processo qualsiasi non intaccano né gli spazi di indirizzamento riservati ad altri processi, né lo spazio di indirizzamento riservato al kernel.

Allo startup del sistema Linux configura la MMU e le strutture dati a supporto della MMU in modo da abilitare la traduzione da indirizzo logico a fisico. Quando questo step è completato, il kernel esegue i suoi compiti nel suo spazio di indirizzamento virtuale.

2.3 Architettura software

Lo sviluppo applicativo e di sistema per piattaforme Linux embedded prevede l'utilizzo di un apposito sistema, detto sistema build, oltre alla piattaforma stessa, detto target. In generale il sistema build è un PC con architettura differente dal target; ad esempio potrebbe essere un PC con architettura x86 o x86-64, mentre il target potrebbe essere un sistema embedded basato su architettura ARM. In figura 2.1 sono mostrati i vari componenti software sia dell'ambiente di sviluppo che della piattaforma embedded.

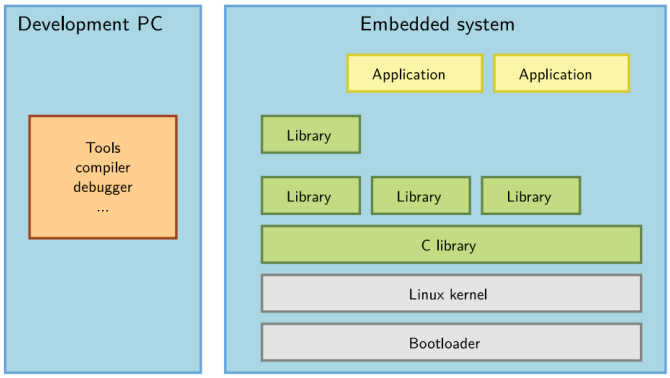


Figura 2.1: Architettura sistema Linux Embedded[10]

2.3.1 Ambiente di sviluppo

L'ambiente di sviluppo contiene la *toolchain di compilazione* che si occupa di generare file binari per uno specifico hardware a partire dal codice sorgente. In generale

una toolchain è costituita da differenti componenti: set di tool, compilatori ed utility (ad esempio il debugger). Il nome toolchain deriva dal fatto che l'otuput generato da ogni componente è usato come input del successivo, questa cascata termina con la generazione del file binario effettuata dall'ultimo componente della toolchain.

In figura 2.2 viene mostrata la differenza tra due diverse varianti di toolchain: nativa e di cross-compilazione. Nel campo dei sistemi embedded si parla spesso di *toolchain di cross-compilazione*, ovvero toolchain installate ed eseguite all'interno un sistema build con una certa architettura (x86, x86-64) che sono in grado di generare binari per target basati su architetture differenti (ARM). Queste toolchain si distinguono dalle cosiddette toolchain native utilizzate nel caso in cui il sistema build e target sono lo stesso sistema.

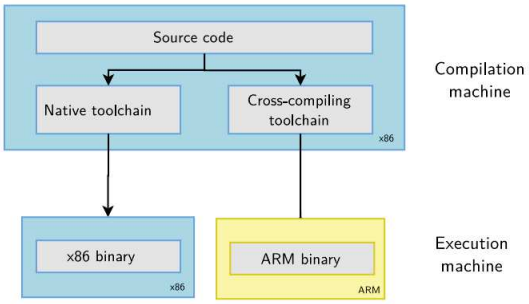


Figura 2.2: Toolchain di cross-compilazione[10]

Una toolchain di cross-compilazione è costituita dai moduli mostrati in figura 2.3.

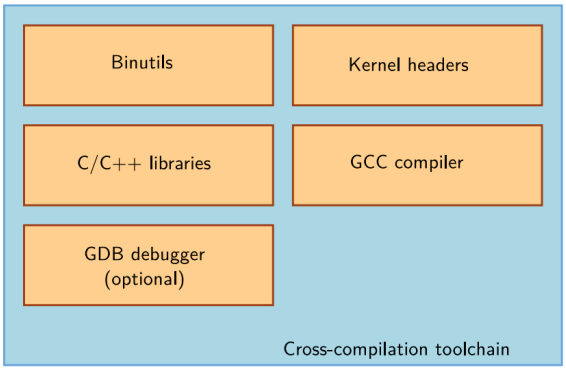


Figura 2.3: Componenti toolchain di cross-compilazione[10]

Ogni modulo si occupa di un preciso compito:

- *Binutils* è un inseme di tool che generano e manipolano binari per una data CPU
- *Kernel headers* costituiscono l'interfaccia tra la libreria C e il kernel stesso. Contengono: dichiarazioni di variabili, costanti, strutture dati e funzioni che devono essere note alla libreria C per interagire con il kernel
- GCC è l'acronimo di *GNU Compiler Collection*, un famoso compilatore free; esso può compilare software scritto in vari linguaggi di programmazione (C, C++, Ada, Fortran, Java) per diverse architetture (ARM, MIPS, PowerPC, x86, x86-64)
- *C library* è un componente essenziale di un sistema Linux, costituisce l'interfaccia tra il kernel è le applicazioni e fornisce un set di API C standard che facilitano lo sviluppo applicativo. Una libreria molto famosa è *glibc*
- GDB è l'acronimo di *GNU Project Debugger*, è un debugger disponibile per molte architetture embedded. Supporta molti linguaggi e può essere usato per controllare il flusso di esecuzione dei programmi. Risulta molto utile per comprendere le cause che portano al crash di un'applicazione.

2.3.2 Bootloader

Il bootloader viene eseguito all'accensione della board ed è responsabile di inizializzare i componenti hardware ed eseguire il kernel. Svolge funzioni avanzate come validare l'immagine del sistema operativo, aggiornarla o scegliere tra diverse immagini basandosi su policy definite dallo sviluppatore.

Si occupa inoltre, come mostrato in figura 2.4, di caricare in RAM il *device tree* che consiste in un file di configurazione della board. Questo file è un database contenente informazioni sui componenti hardware della board ed è usato per inoltrare le informazioni sull'hardware dal bootloader al kernel. Non appena il sistema operativo prende il controllo della board, il bootloader è sovrascritto e cessa di esistere. L'unico modo per eseguirlo nuovamente è effettuare il reset della board stessa.

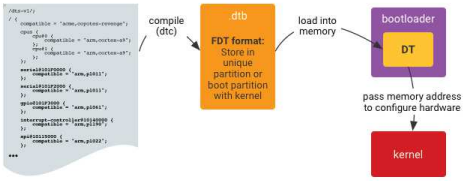


Figura 2.4: Device Tree[19]

Ogni risorsa richiesta dal bootloader deve essere inizializzata e allocata con molta attenzione prima di essere usata. Esempi pratici sono:

- la memorie RAM
- le modalità di passaggio di informazioni tra la memoria non volatile e la RAM
- gestione degli eseguibili. Solitamente un programma scritto in linguaggio C, ha uno stack dentro il quale si ha il contesto di esecuzione di quel programma. In questa fase dato che ancora la RAM non è stata inizializzata non si può creare nessuno stack. L'esecuzione di un qualsiasi programma porterebbe quindi al crash del sistema; ciò significa che il bootloader deve crearsi il proprio contesto di esecuzione.

I progettisti di sistemi Linux embedded devono necessariamente fornire il proprio bootloader, possibilmente apportando le dovute modifiche a partire da uno già esistente. Un bootloader opensource molto diffuso è *U-boot*, usato in varie architetture tra cui ARM, MIPS ed x86.

2.3.3 Kernel Linux

Il kernel Linux gestisce tutte le risorse hardware: CPU, memoria e I/O. Fornisce un set di API che astraggono tali risorse permettendo alle applicazioni utente ed alle librerie di usarle facilmente. Gestisce inoltre gli accessi concorrenti alle varie risorse da parte delle varie applicazioni.

In figura 2.5 sono mostrati i moduli contenuti nel kernel che solitamente sono scritti in linguaggio C e in assembly.

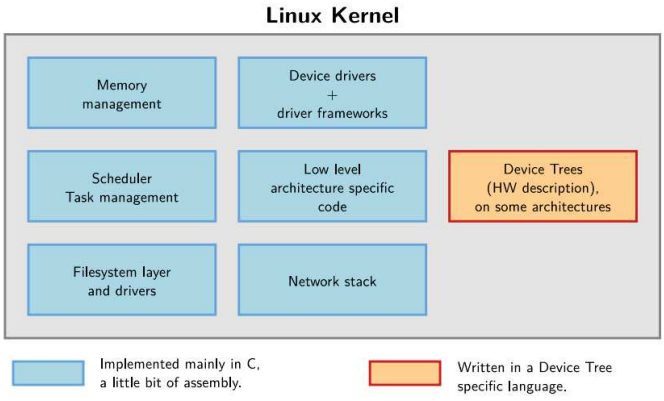


Figura 2.5: Dentro il kernel Linux[10]

Il modulo *Device Trees* è invece scritto in un apposito linguaggio e fornisce una descrizione dell'hardware. Un *Device Tree Source* è compilato in un file binario, il *Device Tree Blob*, e viene fornito al kernel dal bootloader.

Il kernel esegue i suoi compiti nel suo spazio di indirizzamento virtuale. La MMU, come accennato in precedenza, permette di separare processi eseguiti nel contesto kernel da processi eseguiti nel contesto user. L'interfaccia tra il contesto kernel e user è fornito dalle *system call*. Queste funzioni, invocate nello spazio utente, permettono al processore di eseguire istruzioni con tutti i privilegi necessari ad operare nello spazio kernel. Le system call sono incapsulate nella libreria standard C, quindi le applicazioni utente non possono invocarle direttamente, ma devono necessariamente usare le relative funzioni di libreria.

Operazioni svolte dal kernel

Durante l'avvio di un sistema embedded il kernel Linux svolge due compiti di fondamentale importanza. Il processo di avvio di un sistema Linux embedded è schematizzato in figura 2.6.

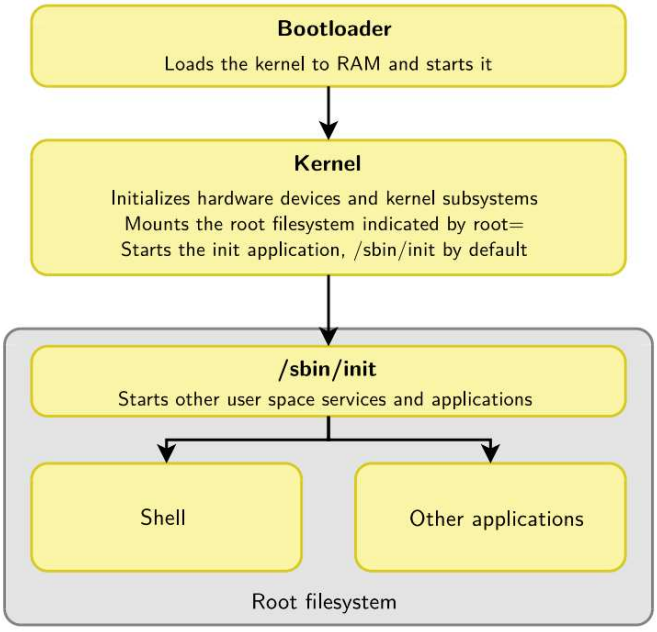


Figura 2.6: Avvio Linux Embedded[10]

Dopo che il bootloader carica il kernel in RAM e lo esegue, esso continua il processo di inizializzazione dei dispositivi hardware; inoltre si occupa di montare il *root filesystem*, un particolare filesystem montato alla base della gerarchia, identificata in Linux dal simbolo /. Un root filesystem è costituito da programmi applicativi e librerie di sistema. Contiene le utility che si occupano di: continuare lo startup del sistema, inizializzare i servizi di rete e console, caricare driver e montare file system addizionali. L'organizzazione dei direttori di un root filesystem Linux è definita dal Filesystem Hierarchy Standard (FHS). Il root filesystem può essere montato a partire da una partizione di: hard disk, chiave USB, SD card, memoria flash, dalla rete (tramite protocollo NFS) o dalla memoria centrale (usando un filesystem precaricato dal bootloader). A prescindere dal dispositivo di caricamento, è necessario avere un root filesystem, altrimenti il kernel entra in stato di *panic* ed interrompe l'avvio della board.

Dopo aver montato il root filesystem il kernel esegue la prima applicazione dello spazio utente, *init*, che ha il compito di completare il processo di inizializzazione del sistema ed avviare tutte le altre applicazioni e servizi dello spazio utente.

2.3.4 Altri componenti

Altri componenti di un sistema Linux embedded sono la libreria C e le varie librerie ed applicazioni utente. La libreria C, come detto, costituisce l'interfaccia tra kernel e librerie/applicazioni utente, che sono quindi costruite basandosi su questa libreria.

Capitolo 3

Studio sistemi di build automatico

Nell'ottica dello sviluppo per piattaforme Linux embedded, partendo da semplici applicazioni fino ad intere distribuzioni, root filesystem, kernel e bootloader, si possono usare i sistem di build; i quali integrano vari componenti software all'interno di un root filesystem effettuando download, estrazione, configurazione, compilazione ed installazione di tali componenti. Tutto ciò può essere fatto:

- manualmente: è la soluzione più complessa in quanto tutto il software deve essere configurato, compilato ed installato nell'ordine corretto, quindi le eventuali dipendenze devono essere risolte manualmente. Librerie e applicazioni devono anche essere riadattate per la cross-compilazione
- tramite l'ausilio di sistemi di build automatico, ovvero tool che automatizzano il processo di build di root filesystem e del kernel. Alcuni permettono anche di effettuare build di tutta la toolchain di cross-compilazione. Questa è la soluzione più vantaggiosa in quanto le dipendenze vengono risolte in automatico
- utilizzando distribuzioni o filesystem esistenti

Prima di entrare nel dettaglio dei sistemi di build automatico, si discutono i meccanismi che stanno alla base della cross compilazione. Innanzitutto un progetto da cross-compilare deve essere correttamente configurato, compilato ed infine installato all'interno del root filesystem. Uno strumento che consente di effettuare questo lavoro è il tool *GNU Make*[11], il cui compito principale è generare eseguibili a

partire da un codice sorgente che può essere scritto in vari linguaggi di programmazione. Solitamente tali sorgenti utilizzano macro, funzioni, costanti e strutture dati definite in altri file. La potenza del tool *make* sta nella risoluzione, trasparente all'utente, di tali dipendenze. Tutto ciò che serve per utilizzare l'utility è un *Makefile*, nel quale sono definite le regole che gestiscono il processo di generazione dell'eseguibile. L'unica difficoltà consiste nella scrittura di un Makefile.

Esistono due tool che generano Makefile in modo relativamente semplice: *autotools*[12] e *cmake*[13]. Gli autotools sono essenzialmente un insieme di tool che compongono un sistema di build basilare. Il funzionamento di questi tool è mostrato in figura 3.1.

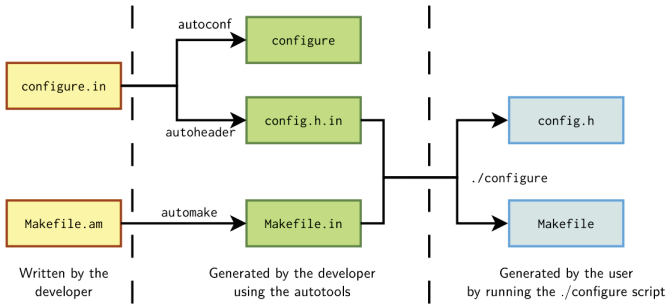


Figura 3.1: Autotools[10]

I tool più importanti sono: *autoconf* ed *automake*. Autoconf, a partire dal file configure.in fornito dallo sviluppatore, genera lo script *configure*. Allo stesso modo, automake, a partire dal file *Makefile.am* fornito sempre dallo sviluppatore, genera un file Makefile.in. Il file configure.in viene utilizzato da un terzo tool, *autoheader*, per generare un file config.h.in. Infine eseguendo lo script *configure*, sono generati un file config.h e il Makefile. L'utility cmake nasce per rimpiazzare automake; per generare un Makefile si basa sull'utilizzo del file CMakeList.txt che contiene un insieme di macro che definiscono le configurazioni per la compilazione del progetto.

Questi sono i meccanismi alla base dei sistemi di build automatico discussi in questo capitolo, nel corso del quale si confronteranno quattro tool opensource con l'obiettivo di individuare il tool che ha le seguenti caratteristiche:

- permette di effettuare build della toolchain di cross-compilazione
- permette di effettuare build del root filesystem
- fornisce tool interni ben integrati tra loro; questa caratteristica incrementa le potenzialità del sistema di build stesso, permettendo all'utente di svolgere

facilmente operazioni complesse. In particolare deve fornire un web front-end che permette all'utente di gestire build da remoto

- facilita lo sviluppo applicativo tramite integrazione in IDE.

3.1 Crosstool-NG

Crosstool-NG[14] è un sistema di build automatico utilizzato solo per build di toolchain di cross-compilazione. Fornisce un'interfaccia per configurare la toolchain prima di effettuare la build automatica. È il successore di *crosstool*, ed è stato progettato per essere:

- facile da utilizzare essendo basato su sistemi di build standard come *autotools* e *make*
- efficiente in quanto utilizza componenti isolati: un file di configurazione, un script di build e un set di patch

3.1.1 Funzionamento

Crosstool-NG è basato principalmente sull'esecuzione dello script *ct-ng*, il cui funzionamento è analogo al tool GNU Make.

Per utilizzare crosstool-NG, si deve innanzitutto configurare, compilare ed installare il tool nel sistema build. In seguito è consigliabile creare una *working directory* che mantiene tutto il lavoro effettuato dal tool. Prima di poter generare la toolchain si devono impostare le seguenti configurazioni: come la toolchain deve essere generata, il path locale dove installarla, l'architettura e lo specifico precessore supportati, e la versione dei componenti software da utilizzare. Tutte queste impostazioni vengono salvate all'interno di un file di configurazione. A questo punto la toolchain può essere generata semplicemente lanciando il seguente comando dalla working directory:

```
ct-ng build
```

L'esecuzione del comando può impiegare diverso tempo, che varia a seconda delle caratteristiche hardware del sistema build.

Una volta generata, la toolchain può essere utilizzata per cross-compilare manualmente qualunque componente software si desidera: da una semplice applicazione, fino ad un root filesystem o un kernel Linux.

Vantaggi:

- genera toolchain di cross-compilazione totalmente custom ed utilizzabile per qualunque modulo software

Svantaggi:

- non genera root filesystem (lo si fa tramite la toolchain non tramite crosstool-NG)
- non è integrato in nessun IDE
- non fornisce un web front-end per gestione delle build da remoto

3.2 PTXdist

PTXdist[15] è un sistema di build sviluppato da Pengutronix nel 2001 che può essere usato per generare sia root filesystem che toolchain di cross-compilazione. Per selezionare e configurare i pacchetti del kernel usa il sistema di configurazione del kernel Linux *kconfig*. Utilizza una collezione di ricette (file che specificano le azioni da compiere per compilare un software) basate su GNU Make o Bash. Opzionalmente sono forniti vari BSP, alcuni generici ed altri specifici a varie piattaforme.

3.2.1 Funzionamento

La generazione di un root filesystem si basa sull'utilizzo del progetto OSLEAS.BSP() e sul comando *ptxdisd*, che deve essere eseguito all'interno di una *working directory*. Innanzitutto si configura, compila ed installa il tool nel sistema build. Quando si utilizza PTXdist per la prima volta si devono impostare ulteriori configurazioni che riguardano: dove conservare i sorgenti e se deve essere usato un proxy per accedere alla rete. Tali impostazioni si configurano tramite il comando:

```
ptxdist setup
```

Prima di generare il root filesystem è necessario avere una toolchain di cross-compilazione. PTXdist permette di utilizzare sia toolchain pre-generate sia appositamente generate. Nel secondo caso la generazione della toolchain viene effettuata a partire da un secondo progetto, OSLEAS.Toolchain().

La procedura di generazione, sia per toolchain che per root filesystem, si effettua con il comando

`ptxdist go`

La sua esecuzione segue sempre lo stesso flusso di operazioni:

- get: si cercano gli archivi dei moduli software da generare. Tali archivi contengono un file di configurazione ed un set di regole che indicano cosa e come viene generato dai sorgenti. All'interno di un archivio potrebbero anche essere contenute le patch da applicare al modulo
- extract: si estrae il contenuto dell'archivio e, se presenti, si applicano delle patch
- prepare: imposta un insieme di configurazioni prima di effettuare la compilazione. Ad esempio si possono impostare il sistema dal quale viene effettuata la build e la piattaforma che eseguirà il codice.
- compile: effettua la cross-compilazione secondo le configurazioni precedentemente impostate
- install: il software compilato viene installato nel path locale al progetto
- targetinstall: viene generato un archivio IPKG che viene inserito all'interno del root filesystem.

Per generare un root filesystem i vari moduli devono essere elaborati seguendo un preciso ordine. PTXdist esegue automaticamente la generazione dei moduli nel corretto ordine e risolve le dipendenze esistenti tra loro.

Vantaggi:

- genera root filesystem
- genera toolchain di cross-compilazione
- esistono BSP sia generici che specifici ad alcune piattaforme

Svantaggi:

- non è integrato a nessun IDE
- non fornisce un web front-end per gestione delle build da remoto

3.3 Buildroot

Buildroot[16] è un tool che semplifica ed automatizza il processo di build di un sistema Linux embedded tramite cross-compilazione. Per ottenere questo risultato buildroot può generare una toolchain di cross-compilazione, un root filesystem, un'immagine del kernel Linux e un bootloader per uno specifico target. Buildroot può utilizzare una qualunque di queste combinazioni, ad esempio si può generare un root filesystem usando una toolchain esistente o una toolchain appositamente generata.

Un grande vantaggio rispetto ai primi due tool è l'integrazione di Buildroot all'interno di Eclipse grazie ad un apposito plug in. Questa caratteristica permette di semplificare la compilazione, l'esecuzione e il debug di applicazioni e librerie. All'interno di Eclipse non vengono però integrate le configurazioni e il processo di build che si effettuano tramite l'utility make. Manca, inoltre, un web front-end che permette di gestire le build da remoto.

3.3.1 Funzionamento

Buildroot è interamente basato su make, quindi il suo utilizzo non prevede ulteriori script come nei due casi precedenti. Analogamente a PTXdist può utilizzare toolchain esterne, ad esempio pre-esistenti o generate tramite crosstool-NG, oppure può generare automaticamente una toolchain di cross-compilazione.

Dopo aver scaricato il progetto Buildroot si configura il sistema di build. Queste impostazioni servono a regolare il processo di build e si effettuano tramite il classico comando di configurazione del kernel Linux, ovvero:

```
make menuconfig
```

Il processo di build viene poi eseguito lanciando semplicemente il comando

```
make
```

Vantaggi:

- genera root filesystem
- genera toolchain di cross-compilazione
- integrazione in Eclipse per la compilazione e debug remoto delle applicazioni e librerie

Svantaggi:

- l'integrazione con Eclipse non fornisce la possibilità di gestire le build
- non fornisce un web front-end per gestione delle build da remoto

3.4 Yocto

Yocto[17] è un progetto open source che fornisce template, tool e metadati che permettono di creare sistemi Linux embedded custom a prescindere dall'architettura utilizzata. È stato fondato nel 2010 da una collaborazione tra vari produttori hardware, produttori di sistemi operativi opensource e compagnie impiegate nel campo dell'elettronica in modo da creare ordine nel mondo dello sviluppo Linux embedded.

Il progetto Yocto fornisce anche degli esempi che ne mostrano le funzionalità. Essi contengono immagini testate dalla comunità Yocto e coprono diversi profili di build per diverse architetture tra cui: ARM, PowerPC, MIPS, x86, x86-64 e sistemi emulati. Il codice di supporto a specifiche piattaforme è fornito tramite appositi layer BSP. Il progetto è integrato in Eclipse tramite un plug-in che permette di sviluppare non solo applicazioni, ma anche intere distribuzioni. Infine esiste un'interfaccia web per il sistema di build chiamata Toaster che permette di gestire varie build anche da remoto.

Yocto è costruito sulle basi di OpenEmbedded (OE)[18], il quale usa il tool di build *BitBake* per generare un'immagine completa di Linux. BitBake e OE sono combinati per formare il reference project di Yocto, chiamato storicamente *Poky*.

Vantaggi:

- genera root filesystem
- genera toolchain di cross-compilazione
- integrazione completa in Eclipse tramite apposito plug-in
- fornisce un web front-end per gestione delle build da remoto
- utilizzabile per la maggior parte delle piattaforme hardware

Svantaggi:

- date le numerose funzionalità richiede molto tempo per un apprendimento completo

3.4.1 Metadati

I metadati utilizzati dal progetto Yocto sono: ricette, file di configurazione, classi, e file append.

Ricette

Le ricette sono i metadati alla base di BitBake e per ogni componente software forniscono le seguenti informazioni:

- informazioni descrittive sul pacchetto (autore, homepage, licenza e così via)
- versione della ricetta
- dipendenze esistenti
- locazione del codice sorgente e come recuperarlo
- patch del codice sorgente richieste, dove trovarle e come applicarle
- impostazioni di configurazione e compilazione del codice sorgente
- path del target dove sarà salvato il pacchetto (o i pacchetti) creato

Nel contesto di BitBake, o di qualsiasi tool che utilizza BitBake come sistema di cross-compilazione, i file con estensione .bb sono file ricetta.

File di configurazione

I file di configurazione contengono i metadati che definiscono come effettuare le build. Questi file, che hanno estensione .conf, definiscono opzioni di configurazione della macchina, della distribuzione, del compilatore, le configurazioni generali e dell'utente. Permettono di impostare il target per il quale si vuole creare l'immagine, dove salvare i sorgenti scaricati ed altre particolari configurazioni.

Gli script oe-init-build-env o oe-init-build-env-memres inizializzano l'ambiente di build utilizzando le configurazioni di default di questi file. In più il secondo script, inizializza BitBake come programma server memory-resident. Questi script, oltre a settare delle variabili di ambiente, creano la *Build Directory* se non è già esistente. Su questa directory viene salvato tutto il lavoro effettuato durante la fase di build del software.

Classi

I file classe, che hanno estensione .bbclass, contengono funzionalità comuni che possono essere condivise tra varie ricette all'interno della distribuzione. Quando una ricetta eredita una classe, ne eredita anche le impostazioni e le funzioni. Il source tree di BitBake attualmente viene fornito con una classe chiamata base.bbclass la quale è sempre inclusa automaticamente per tutte le ricette e le classi. Questa classe contiene le definizioni per i task più comuni: fetch ed estrazione ricette, configurazione, compilazione, installazione e packaging. Queste attività vengono spesso sovrascritte o estese da altre classi aggiunte durante il processo di sviluppo del progetto.

File append

I file append, con estensione .bbappend, estendono o sovrascrivono le informazioni per una ricetta esistente. BitBake si aspetta che ogni file append corrisponda ad una ricetta e che condividono parte del nome. I nomi dei file ricetta ed append possono differire solo nell'estensione utilizzata, ad esempio formfactor_0.0.bb è la ricetta, mentre formfactor_0.0.bbappend è il relativo file append.

OpenEmbedded Core

OpenEmbedded Core contiene un layer di ricette, le classi ed un insieme di file associati, comuni a tutti i sistemi basati su OE, incluso Yocto. Questo set di metadati è infatti mantenuto sia dal progetto Yocto che dal progetto OpenEmbedded. Prima della versione 1.0, Yocto convogliava ricette ed altri metadati dal progetto OE all'interno del sistema di riferimento *Poky*. In seguito Yocto ed OE si accordarono sulla condivisione di un set comune di metadati, *oe-core*, il quale contiene la maggior parte delle funzionalità presenti in *Poky*.

3.4.2 Architettura

L'ambiente di sviluppo[20] del progetto Yocto, come mostrato in figura 3.2, è formato da varie aree funzionali.

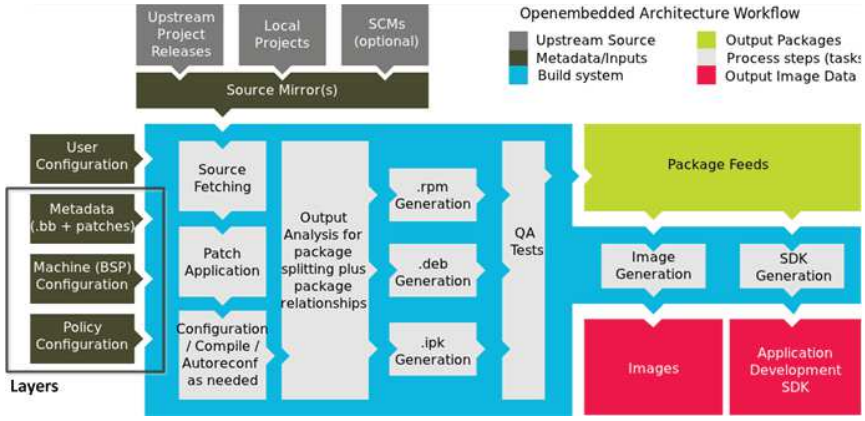


Figura 3.2: Architettura Yocto[20]

Layer

I layer consentono di separare i metadati differenziandoli per: software, informazioni sull'hardware, metadati inerenti la distribuzione e le policy adottate. Tutti i layer hanno una struttura ben definita e contengono un file di licenza se il layer deve essere distribuito. In questo caso è buona norma avere anche un file README contenente le istruzioni generali. All'interno di un layer sono inoltre contenute le directory:

- conf, una directory contenente file di configurazione specifici per il layer
- recipes-*, uno o più directory contenenti ricette.

Per illustrare come utilizzare i layer per mantenere la modularità, si consideri l'esempio di ricette a supporto di uno specifico target, che abitualmente risiedono in un layer BSP. In questo scenario, tali ricette dovrebbero essere isolate da altre ricette e metadati che supportano, ad esempio, un nuovo ambiente Graphical User Interface (GUI). Si avrebbero quindi un paio di layer: uno per le configurazioni della macchina e uno per l'ambiente GUI. Ciò permetterebbe per una specifica macchina di aggiungere caratteristiche particolari della GUI all'interno del layer BSP, senza intaccare le ricette interne al layer GUI stesso. Tutto ciò è possibile tramite file *append*.

In generale esistono tre tipi di layer:

- Policy Configuration: fornisce le configurazioni delle policy per la propria distribuzione. È consigliabile separare i vari tipi di configurazioni all'interno

di un proprio layer. Le impostazioni fornite nei file di configurazione del layer sovrascrivono quelle fornite dal file di configurazione generale.

- Machine Configuration: contiene layer BSP, i quali forniscono le informazioni sulla configurazione di una particolare architettura

- Metadata: software layer contenti ricette, patch e append file forniti dall'utente. In questo layer non sono inclusi metadati specifici alla particolare distribuzione o alla particolare architettura.

Source file

Per cross-compilare un qualsiasi modulo software, sia esso una distribuzione o un'applicazione, OE deve avere accesso ai vari file sorgente. Questi ultimi possono essere reperiti da tre diverse aree di upstream, mostrate in figura 3.2:

- Upstream Project Releases: i sorgenti sono contenuti in una certa locazione in forma di file archivio, i quali contengono ricette per un particolare modulo

- Local Projects: in questo caso i sorgenti sono contenuti in un certo path locale. Il metodo canonico per includere un progetto locale all'interno di una build si basa sull'utilizzo della classe externalsrc nella relativa ricetta. Questa classe supporta la compilazione di moduli software a partire da sorgenti non mantenuti nei repository OpenEmbedded. Infine si modifica il file di configurazione generale, local.conf in modo tale che BitBake effettui la cross-compilazione della ricetta

- Source Control Managers (SCM), opzionale. Una fonte alternativa da cui il sistema di build può ottenere i file sorgenti è attraverso un SCM come Git o Subversion. In questo caso un repository è clonato o ne viene fatto il checkout, cioè l'estrazione dei dati dal repository stesso.

Una ulteriore area che svolge un ruolo fondamentale nel reperire i sorgenti è il *Source Mirror(s)*. Esistono due tipi di mirror: *premirrors* e *regular mirrors*. Inizialmente i sorgenti vengono cercati nell'area di premirror, successivamente vengono ispezionate le aree di upstream. I premirror sono appropriati quando si ha una cartella condivisa che è locale ad una certa organizzazione. I regular mirror sono rappresentati invece da una risorsa di rete alternativa alla risorsa principale, nel caso in cui quest'ultima non dovesse essere raggiungibile.

Questi file vengono salvati in un direttorio che prende il nome di *Source Directory*.

Package feeds

In questa area sono contenuti i pacchetti generati dal sistema di build che in seguito saranno utilizzati per generare immagini del sistema operativo o Software Development Kit (SDK). Il formato del package da generare è configurabile, si può scegliere tra: RPM, DEB, IPK o TAR. Quest'area può anche essere copiata e condivisa tramite un web server in modo da facilitare l'estensione o l'aggiornamento di immagini esistenti a runtime.

Sistema di build (BitBake)

Il macroblocco build system di figura 3.2 è il cuore del progetto Yocto. Al suo interno sono contenuti vari processi sotto il controllo di BitBake, un tool scritto in Python. Esso è responsabile di effettuare il parse dei metadati, dai quali ricava la lista di task da eseguire. Analogamente all'utility GNU Make, BitBake controlla il processo di build del software. GNU Make effettua tale controllo tramite Makefile, mentre BitBake utilizza le ricette, inoltre permette la definizione di task più complessi come l'assemblaggio di intere distribuzioni Linux.

I task sono eseguiti da BitBake rispettando un certo ordine:

- fetch dei sorgenti. Come detto i sorgenti sono disponibili sotto forma di pacchetti che contengono le ricette. La prima fase consiste quindi nel fare fetch del pacchetto, dal quale viene estratta la ricetta corrispondente. I task do_fetch e do_unpack eseguono rispettivamente queste due azioni
- applicazione patch. Durante questa fase BitBake cerca e, se esistono, applica le patch alla ricetta tramite il task do_patch
- configurazione e compilazione. Questo step si occupa di configurare e cross-compilare i sorgenti. Può essere considerato lo step centrale di tutto il processo. Bitbake esegue questi compiti tramite i task do_configure e do_compile. L'output della cross-compilazione viene inoltre salvato all'interno della Build Directory tramite il task do_install
- split package. In questo step BitBake analizza l'output generato dal processo di cross-compilazione e divide tali file in package tramite i task do_package e do_packagedata. Possono essere generati vari formati di package: RPM, DEB e IPK. Successivamente questi package sono sottoposti ad un ulteriore controllo di qualità prima di essere salvati nell'area di Package Feeds usando rispettivamente i task: do_package_write_rpm, do_package_write_deb, do_package_write_ipk

- generazione immagini. I pacchetti dell'area Packege Feeds vengono usati in questo step per generare l'immagine del root filesystem. La generazione di questa immagine è divisa in sottotask. Inizialmente viene eseguito il task do_rootfs che crea il root filesystem. Il task successivo è do_image che inizia il processo di generazione dell'immagine. Questo task crea dinamicamente altri task do_image_* che si occupano di comprimere l'immagine del root filesystem. La generazione dell'immagine (o set di immagini) è termina con il task do_image_complete
- generazione SDK . Yocto è in grado di generare anche SDK tramite l'esecuzione di script di installazione generati durante questo step. I task che generano gli SDK si distinguono in: do_populate_sdk (per la versione standard) e do_populate_sdk_ext (per la versione estesa). La SDK standard fornisce la toolchain di cross-development e librerie custom per una certa immagine. La SDK estesa rispetto alla versione standard fornisce anche un sistema di build interno e la possibilità di utilizzare devtool[1].

Per ognuno dei task completati con successo, BitBake scrive un file *stamp* all'interno della Build Directory. Questi file sono usati per determinare se un task deve essere rieseguito. Il meccanismo alla base di questo controllo è il calcolo del checksum degli input al task stesso, che viene poi incluso nel nome del file stamp. In seguito BitBake per discriminare se un task è stato eseguito, calcola il ckecksum degli input e verifica l'esistenza di un file stamp che abbia quel checksum. Se non viene trovata nessuna corrispondenza il task viene rieseguito.

L'esecuzione di un task consiste in una sequenza di operazioni. Normalmente BitBake deve generare tutte le informazioni necessarie allo svolgimento di quel particolare task. Esiste la possibilità di saltare alcune fasi dei task utilizzando degli oggetti precostruiti, messi a disposizione del processo tramite la *shared state cache (sstate)*. Questo comportamento viene attivato usando una variante particolare del task stesso, nota come variante *setscene*.

Immagini

Le immagini prodotte da OE sono delle forme compresse del root filesystem pronte ad essere installate sul target. L'output di BitBake, in parte, è formato da diversi tipi di immagini salvate nella Build Directory. Si distinguono:

- kernel-image

[1] devtool permette di effettuare build di progetti il cui output sarà una porzione del root filesystem costruito tramite BitBake

- root-filesystem-image
- bootloaders
- kernel-modules: tarball che contengono tutti i moduli generati per il kernel
- symlinks: un link simbolico usato per puntare alla build più recente per ogni macchina. Questi link possono essere utili per script esterni che cercano di ottenere l'ultima versione di ogni file.

SDK

Dal processo di generazione dell'SDK si possono ottenere una SDK standard o una extensible SDK. In entrambi i casi, in output si ottiene un script di installazione dell'SDK, che installa: toolchain di cross-development, un set di librerie ed header file, e genera uno script di setup dell'ambiente. La toolchain può essere considerata come parte del sistema build, mentre le librerie e gli header come parte target poiché sono generati per il target hardware. Lo script di setup dell'ambiente deve essere eseguito prima di utilizzare la toolchain di cross-development.

3.4.3 Tools

Per poter generare una distribuzione Linux embedded, Yocto si affida a diversi tool correlati tra loro. In tal modo si aumenta l'interoperabilità e l'usabilità di Yocto stesso.

Bitbake

Come detto BitBake è il tool che esegue i task su cui si basa il sistema di build OpenEmbedded. Prima di eseguire BitBake si devono impostare le variabili di ambiente tramite l'esecuzione di uno dei seguenti due script:

- `oe-init-build-env`
- `oe-init-build-env-memres`

In seguito si può eseguire il tool BitBake digitando direttamente da terminale il comando `bitbake <recipe_to_build>`. Il processo può durare da pochi minuti a diverse ore, dipende dal tipo di ricetta (se è una semplice applicazione o una distribuzione) e dalle caratteristiche dell'hardware della macchina host.

Plug-in Eclipse

Il plug-in Eclipse[21] permette di utilizzare l'IDE come strumento di sviluppo applicativo, ma anche di sistema.

Fino alla versione 2.0 di Yocto lo sviluppo applicativo era effettuato tramite l'utilizzo di Application Development Toolkit (ADT) e toolchain di cross-compilazione stand-alone. Dalla versione 2.1 è stata effettuata una transizione da questi strumenti alle più tradizionali SDK, che contengono:

- cross-development toolchain, che includono compilatore, debugger e altri strumenti di supporto
- librerie, file header e di definizione simboli che sono specifici ad una certa immagine
- script di setup dell'ambiente SDK, un file .sh che una volta eseguito imposta l'ambiente di cross-development settando delle variabili di ambiente

L'ambiente di sviluppo applicativo fornito dalla SDK può essere utilizzato all'interno dell'IDE Eclipse grazie ad un apposito plug-in che permette quindi di utilizzare Yocto all'interno dell'IDE stesso. Il vantaggio di integrare Yocto in Eclipse consiste nella possibilità di sfruttare gli strumenti messi a disposizione dell'IDE, come l'autocompletamento del codice, facilitando di molto la fase di sviluppo.

Esiste anche un'estensione di Eclipse, *Yocto Project BitBake Commander*, che permette di creare metadati da inserire all'interno del progetto Yocto direttamente dall'IDE. Tale estensione permette anche di modificare layer e ricette.

In generale il modello si sviluppo tramite SDK prevede l'utilizzo di una macchina apposita usata per: sviluppare applicazioni, immagini di sistemi operativi e kernel. Tale macchina è concettualmente slegata dalla macchina che contiene Yocto (il sistema build), quindi è possibile sviluppare e testare indipendentemente qualunque codice. Una volta ottenuto il prodotto desiderato lo si può integrare all'interno di un'immagine usando il sistema build, il quale avendo Yocto al suo interno genera nuovamente l'immagine applicando la modifica desiderata.

Toaster

Toaster[22] è un'interfaccia web verso il sistema di build Yocto. Tramite Toaster si possono configurare build custom ed eseguirle, inoltre fornisce strumenti di statistica ed analisi sul processo di build. Permette anche di:

- selezionare ed effettuare build di qualunque layer presente in OE anche su server remoti

- importare layer custom ed eseguire le build di tali layer
- aggiungere e rimuovere qualunque layer
- impostare delle specifiche variabili di configurazione che guidano il processo di build
- selezionare uno o più target hardware
- visualizzare la struttura delle directory dell'immagine della distribuzione
- visualizzare il valore delle variabili di configurazione impostate per effettuare la build

Le analisi e statistiche che Toaster permette di effettuare, forniscono ulteriori informazioni su:

- processo di build, dando la possibilità di analizzare cosa è stato generato ed installato nella distribuzione finale in termini di ricette e pacchetti. Tali informazioni sono collezionate e conservate in un database
- eventuali errori che possono avvenire durante il processo di build, tramite appositi messaggi di warning e di errore utili in fase di debug
- task eseguiti da BitBake ed informazioni sulle relazioni tra ricette, pacchetti e task, eventualmente permette di riutilizzare i risultati di tali task in altre build
- tempo impiegato per effettuare una build completa, per eseguire un singolo task, utilizzo della CPU e del disco

Le build eseguite tramite Toaster sono organizzate in progetti, quando si crea un progetto si richiede di selezionare una release di Yocto oppure una versione esistente nel sistema locale. Toaster permette inoltre di tenere traccia di build eseguite da linea di comando che prendono il nome di *Command line builds*. Per poter raccogliere le informazioni su questo tipo di progetto è necessario inizializzare Toaster, tramite script, prima di iniziare il processo di build.

Toaster è stato progettato per lavorare utilizzando il framework Django[2][23]. In generale questo framework effettua un mapping diretto tra le URL e funzioni, tramite file urls.py. Un secondo mapping è fatto tra le URL e le pagine visualizzate

[2]Django è un framework Web di Python ad alto livello che incoraggia lo sviluppo rapido e un design pulito e pragmatico. Si occupa di gran parte della gestione delle problematiche dello sviluppo Web, permettendo al programmatore di concentrarsi sulla scrittura dell'applicazione. È free e open source.

dal client, tramite file views.py. Grazie a questi due mapping si associano le funzioni, in questo contesto di BitBake, alle pagine. Le funzioni eseguono quindi le applicazioni lato server e possono restituire:

- un template Toaster, i quali sono localizzati all'interno del progetto Yocto
- una qualunque risposta HTTP contenente ad esempio un documento JSON

Toaster memorizza le informazioni in un database completamente astratto dall'*object relation mapping (orm)* di Django. In questo modo non è necessario considerare dettagli come la gestione delle connessioni, la scelta del database ed il mapping dei dati in oggetti. Lo schema del database Toaster viene definito nel file models.py anche esso contenuto nel progetto Yocto.

Il setup di Toaster può essere fatto seguendo due diverse modalità: istanza locale o servizio condiviso.

La prima modalità richiede che tutti i componenti risiedano in un singolo host come si vede in figura 3.3. Fondamentalmente un'istanza locale è adatta per lo sviluppo effettuato da un singolo utente all'interno di un unico sistema di sviluppo.

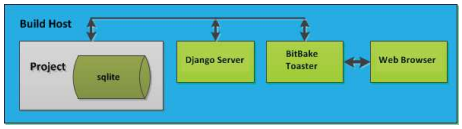


Figura 3.3: Istanza locale[22]

Il setup di Toaster come servizio condiviso è invece adatto allo sviluppo effettuato da parte di più utenti, ad esempio all'interno di una rete aziendale. Lato server, i servizi offerti da Toaster possono essere divisi in un sistema distribuito di più host. Una situazione tipica di servizio condiviso è mostrata in figura 3.4. In questo scenario si hanno tre host, contenenti rispettivamente:

- progetti, BitBake e Toaster
- database
- il web server con il quale i vari client possono interfacciarsi. Nel caso di Toaseter il web server utilizzato è Apache[24]

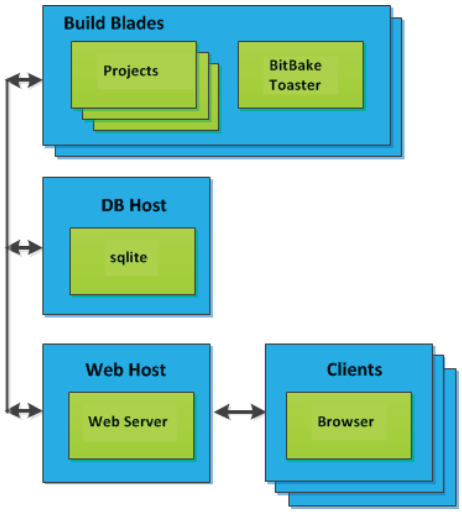


Figura 3.4: Servizio condiviso[22]

3.4.4 Task di uso generale

Il progetto Yocto fornisce un insieme di procedure da utilizzare durante la fase di sviluppo; due di queste riguardano come creare ed integrare nuovi layer e ricette all'interno di Yocto [26]. Esiste inoltre una procedura per creare immagini di distribuzioni custom [26].

Creare layer

OE supporta l'organizzazione di metadati in layer, i quali permettono di creare un isolamento tra le varie personalizzazioni. All'interno del progetto Yocto sono già inclusi layer di vario genere, riconoscibili in quanto il nome di tali directory inizia con la stringa meta-.

Il progetto Yocto fornisce inoltre degli script che automatizzano il processo di creazione di nuovi layer. Nel caso di layer BSP si utilizza lo script *yocto-bsp*, mentre per layer generici lo script *yocto-layer*. Quando si crea un nuovo layer ci si deve assicurare che non ne esista già uno contenente gli stessi metadati, in questo caso ovviamente avere un nuovo layer è del tutto superfluo. Successivamente si crea una directory il cui nome inizia con la stringa meta- seguita dal nome del layer; questa è una pratica consigliata, ma non obbligatoria.

All'interno della directory appena creata deve essere presente un file, con percorso conf/local.conf, contenente le configurazioni locali al layer. Se il layer è di supporto per un certo target, le configurazioni della macchina saranno collezionate all'interno dal percorso conf/machine; se di supporto alla distribuzione all'interno di conf/distro; se nel layer sono presenti nuove ricette allora saranno contenute in directory del tipo recipes-*.

Fatto ciò si deve istruire OE dell'esistenza del nuovo layer modificando il file conf/local.conf all'interno della Build Directory. In questo file si ha una variabile BBLAYER che punta un insieme di layer, a questi si va ad aggiungere il percorso del nuovo layer. In questo modo OE conosce la locazione del layer e di tutti i metadati contenuti al suo interno, tenendone conto durante il processo di build.

Un concetto importante è l'associazione di una priorità assegnata ai layer, molto utile nel caso in cui diversi layer contengono ricette, o file di append, con lo stesso nome. In questi casi la precedenza viene assegnata alla ricetta, o la file di append, contenuta nel layer con maggiore valore di priorità.

Nuove ricette

Le ricette sono di fondamentale importanza all'interno di Yocto in quanto ogni componente software è costruito a partire da una ricetta. Queste ultime possono essere create:

- da zero; è la soluzione più dispendiosa da un punto di vista del lavoro da effettuare
- usando *recipetool*, un tool fornito da Yocto che permette di creare nuove ricette in modo automatico, partendo dal codice sorgente già esistente. Una volta creata la ricetta la si deve inserire all'interno della directory contente i metadati del layer e darle un nome appropriato.
- da ricette già esistenti apportando le dovute modifiche

In figura 3.5 è mostrato il flusso seguito durante la creazione di una nuova ricetta, alcuni di questi step sono già stati descritti nella sezione 3.4.2. I passi aggiuntivi sono:

- configurazione: molti software forniscono mezzi per impostare opzioni di configurazione prima della compilazione, tipicamente tramite script o modificando un file di configurazione della build. Se la compilazione viene fatta usando Autotools o CMake, le configurazioni sono mantenute rispettivamente nei file configure.ac e CMakeList.txt. Se non si utilizzano Autotools o CMake si deve necessariamente implementare il task do_configure
- installazione: questa fase dipende da come viene fatto il build del software. Nel caso in cui si usa Autotools o CMake, l'installazione viene eseguita senza uno specifico task do_install; in caso contrario, si deve specificare tale task ed eseguire il comando make install
- abilitare i system services: se occorre installare un service, che solitamente è un processo che si avvia al boot ed eseguito in background, si devono includere ulteriori definizioni nella ricetta. Tale funzione può essere compiuta tramite il task do_install_append o tramite uno script di inizializzazione del service stesso. OE abilita l'avvio di service tramite SysVinit o systemd. Entrambi sono dei service manager usati dal programma init per controllare funzioni base del sistema; systemd è il manager più aggiornato ed offre dei miglioramenti in termini di gestione rispetto a SysVinit
- script post installazione: questi script vengono eseguiti immediatamente dopo la fase di installazione o durante l'inclusione del package che rappresenta il modulo software all'interno di una immagine

- test: i test del software ottenuto dopo la costruzione vengono effettuati direttamente sul target.

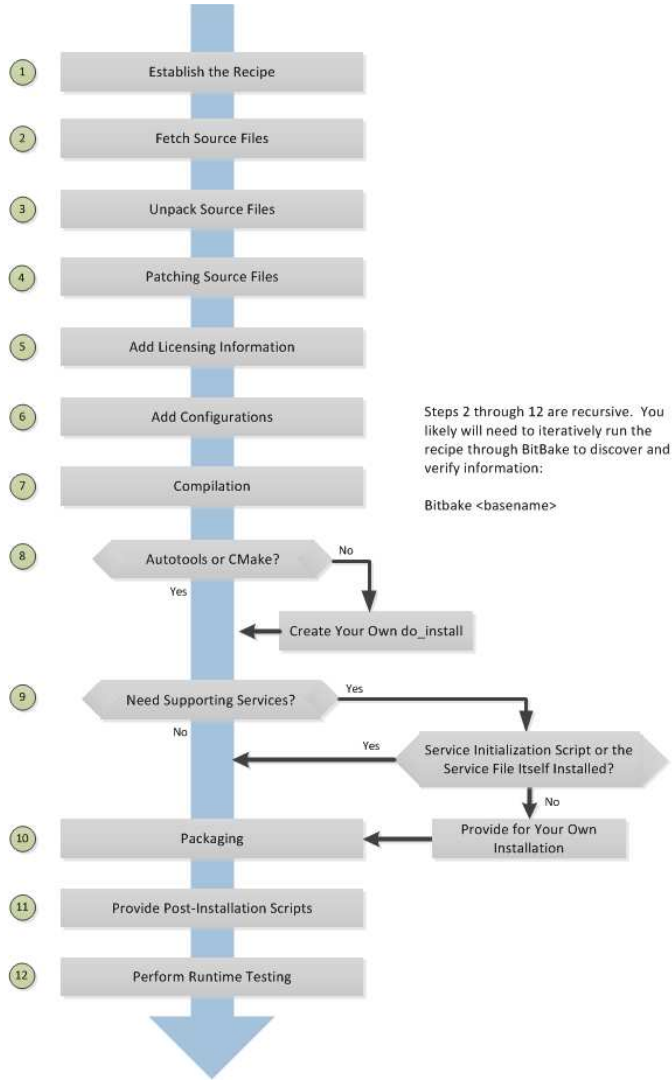


Figura 3.5: Flusso generazione ricetta[26]

Immagini custom

Una delle caratteristiche che rende Yocto uno strumento molto potente e flessibile è la possibilità di poter creare immagini custom, questo può essere fatto in tre modalità differenti.

La prima modalità è la più semplice, ma è limitata ad un uso locale. Consiste nell'aggiunta di un package nel file local.conf tramite la variabile IMAGE_INSTALL usata con l'operatore append come nell'esempio:

```
IMAGE_INSTALL_append = " strace"
```

Il comportamento dell'operatore append è quello di concatenare il contenuto alla variabile in questione, in questo caso IMAGE_INSTALL. È importante inserire anche lo spazio iniziale in quanto l'operatore non lo aggiunge di default. Basandosi sull'utilizzo di variabili locali tale soluzione è la meno flessibile in quanto il suo effetto si ha su tutte le build e conseguentemente su tutte le immagini.

Una seconda alternativa si basa sull'abilitazione e disabilitazione di caratteristiche ad alto livello delle immagini tramite le variabili IMAGE_FEATURES ed EXTRA_IMAGE_FEATURES. In particolare è raccomandato modificare la prima variabile nella ricetta dell'immagine, mentre la seconda nel file local.conf, in entrambi i casi il comportamento è analogo. Nel caso si usi la variabile IMAGE_FEATURES il sistema di build aggiunge automaticamente tali caratteristiche alla variabile IMAGE_INSTALL specificata nella ricetta di quella particolare immagine. Rispetto alla soluzione precedente si stanno quindi aggiungendo nuove caratteristiche estendendo la ricetta, o creandone una opportunamente, piuttosto che modificando le configurazioni locali. Nel caso si usi la variabile EXTRA_IMAGE_FEATURES si va a modificare nuovamente il file di configurazione locale local.conf, in seguito il valore di tale variabile viene aggiunto al valore della variabile IMAGE_FEATURES contenuto nel file meta/conf/bitbake.conf.

La terza soluzione prevede la scrittura di una ricetta totalmente personalizzata. In questo contesto, ovviamente, si ha il pieno controllo delle caratteristiche che si vogliono aggiungere all'immagine.

3.5 Confronto

Yocto si distingue rispetto agli altri sistemi di build automatici. In figura 3.6 viene mostrata una tabella che riassume le caratteristiche dei sistemi visti.

Tool	OE/Yocto	Buildroot	PTXdist	crosstool-NG
Build toolchain	✓	✓	✓	✓
Build root filesystem	✓	✓	✓	
IDE Integration	✓	✓		
Web front-end	✓			

Figura 3.6: Confronto generatori toolchain di cross-compilazione

Yocto risulta essere la scelta migliore in quanto:

- può generare toolchain di cross-compilazione
- può generare il root filesystem
- esistono dei plug-in per Eclipse che integrano Yocto all'interno dell'IDE
- offre un web front-end che permette di gestire la generazione di distribuzioni anche da remoto

Gli altri generatori di toolchain offrono solo un sotto insieme delle funzioni fornite da Yocto, tra questi l'unica alternativa che sembrerebbe attendibile è Buildroot.

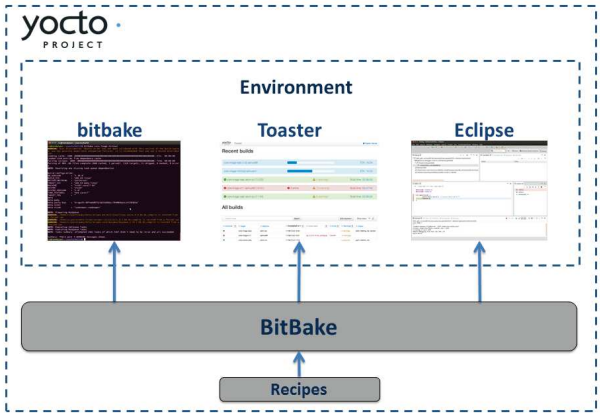


Figura 3.7: Yocto

Il grande vantaggio di Yocto rispetto ai competitor non è però legato solo alle funzionalità offerte, ma al fatto che queste cooperano tra loro, come mostrato in figura 3.7. Riassumendo, Yocto per cross-compilare moduli software, si basa su file che prendono il nome di *recipe* o *ricette*. Questi file possono essere organizzati in *layer*, che permettono di separare le ricette in base alle loro funzionalità e di selezionare il gruppo di ricette da cross-compilare durante il processo di build. L'insieme di ricette e layer è gestito da una gerarchia di file di configurazione, i quali regolano l'intero processo

di cross-compilazione. Le ricette vengono elaborate dal componente principale del progetto Yocto, BitBake, che costituisce il back-end. Esso è un generatore di file archivio che a fronte di una ricetta in input compila il modulo software desiderato per il target hardware.

Yocto fornisce tre diversi ambienti in grado di interfacciarsi verso il back-end: bitbake, Toaster ed Eclipse. Il primo è un tool utilizzabile da riga di comando che mostra le varie operazioni compiute dal back-end durante l'elaborazione di una ricetta. Toaster offre un'interfaccia web dalla quale è possibile: creare progetti, cross-compilare ricette e monitorare l'esito del processo. Il setup di Toaster può essere fatto in modo da eseguirlo come istanza locale oppure come servizio condiviso. La prima modalità è adatta per lo sviluppo effettuato da un singolo utente all'interno di un unico sistema di sviluppo. Il servizio condiviso è adatto al caso di una comunità di sviluppatori che utilizzano un sistema distribuito contenente i vari componenti di Toaster. Infine esistono dei plug-in di Eclipse che permettono di integrare Yocto all'interno dell'IDE. In questo modo per ogni progetto Yocto si possono usare tutti gli strumenti forniti dall'IDE stesso.

Capitolo 4

Analisi piattaforme hardware e software

4.1 Hardware

L'hardware utilizzato per lo svolgimento dei test del tool Yocto e di generazione della distribuzione software richiesta è la board commerciale Udoo Quad Core. La scelta di questa board è legata alla compatibilità hardware con la board proprietaria Magneti Marelli.

Udoo Quad Core

Udoo è un progetto nato da SECO USA Inc e Adilab.srl, in collaborazione con un team di ricerca multidisciplinare. La board Udoo Quad Core, figura 4.1, ha le seguenti caratteristiche:

- processore NXP i.MX 6 ARM Cortex-A9 CPU Quad core da 1GHz (integra funzionalità di grafica)
- processore Atmel SAM3X8E ARM Cortex-M3 CPU
- RAM DDR3 da 1GB
- 76 fully available GPIO
- pinout compatibile con Arduino Uno R3

- HDMI e LVDS + Touch (segnali I2C)
- Ethernet RJ45 (10/100/1000 MBit)
- modulo WiFi
- Mini USB e Mini USB OTG
- USB tipo A (x2) e connettore USB
- Analog Audio e Mic
- SATA
- connessione Camera
- slot Micro SD che funge da dispositivo di boot
- Tensione di alimentazione 12V e connettore batteria esterna

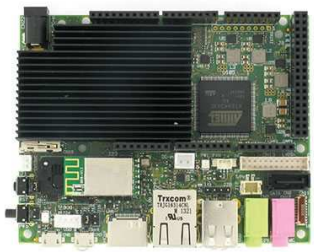


Figura 4.1: Udoo Quad[8]

Board proprietaria Magneti Marelli

Questa board è stata progettata per comunicazioni V2X che si basa sullo standard IEEE 802.11p.

- processore: iMX6 Solo, iMX6 Dual, iMX6 Quad, iMX6 Quad Plus
- RAM DDR3 da 1GB
- flash NAND da 512 MB
- flash eMMC da 32 GB
- HSM
- GPS
- automotive ethernet
- CAN High Speed & Low Speed
- sensori: accelerometro, giroscopio
- connettività: Wi-Fi, bluetooth, 4G
- connectività V2X: Autotalks, NXP, Qualcomm, Renesas, Murata

Figura 4.2: Board Magneti Marelli

4.2 Framework V2X Magneti Marelli

Il framework è stato progettato con l'obiettivo di far coesistere l'insieme degli standard per la comunicazione interveicolare. In particolare si considerano gli standard:

Europeo (ETSI), Americano (WAVE) e Giapponese (ARIB). Come mostrato in figura 4.3 il framework è diviso in vari livelli; un livello hardware (lo strato alla base) sopra il quale sono stati implementati vari livelli software.

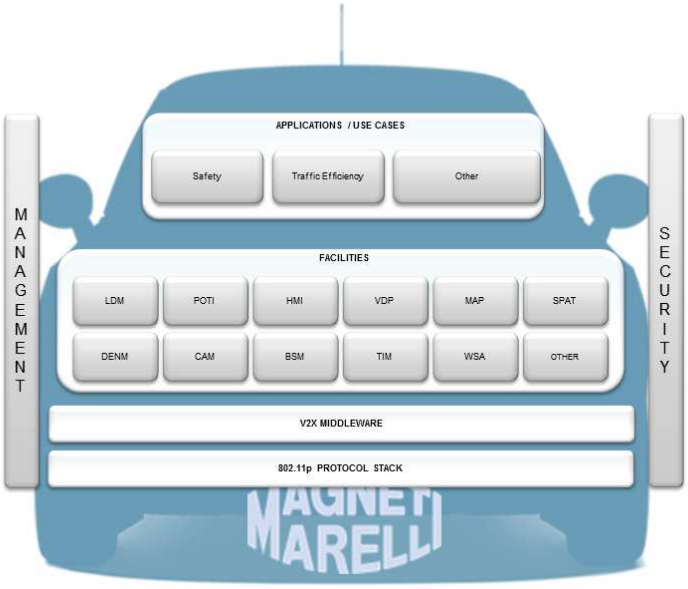


Figura 4.3: Framework V2X Magneti Marelli

4.2.1 Protocol Stack

È lo strato rappresentato dal blocco 802.11p PROTOCOL STACK presente in figura 4.3. Gli standard V2X WAVE ed ETSI sono basati sul protocollo 802.11p[28], ovvero comunicazioni wireless a corto raggio, note anche con il termine inglese Dedicated Short-Range Communications (DSRC). La banda allocata per 802.11p ha larghezza 75 MHZ, con portante a 5.9 GHz. ARIB è sempre basato su tecnologie 802.11p, ma la frequenza di trasmissione ha portante a 700 MHz.

4.2.2 Middleware

Il middleware interfaccia i livelli facilities ed application con il *driver stack V2X*. Quest'ultimo viene fornito da aziende produttrici di software, in conformità ai

diversi standard. Il compito del middleware è rendere indipendenti ed adattabili i livelli superiori rispetto al particolare stack V2X in uso.

All'interno del middleware è presente il modulo Sender&Receiver (S&R) che principalmente smista messaggi, basandosi sull'header, sia in trasmissione che ricezione; quindi da e verso le facilities.

4.2.3 Facilities

Il layer facilities è costituito da un insieme di moduli software, i quali svolgono diverse mansioni. Tra questi, alcuni effettuano comunicazione bidirezionale con il S&R, mentre altri comunicano soltanto in ricezione. Questi moduli possono essere raggruppati in base allo standard in cui sono definiti. Al momento sono considerati i moduli definiti negli standard WAVE ed ETSI, ed i moduli comuni ai due standard. Si ricorda che l'obiettivo del framework V2X è proprio quello di far coesistere tutti questi standard.

I moduli comuni ai due standard sono:

- Local Dynamic Map (LDM): è il database di una stazione ITS contenente le informazioni necessarie al funzionamento delle applicazioni contenute in quella particolare stazione ITS. I dati possono essere ricevuti da diverse tipologie di sorgenti, come: altri veicoli, unità presenti nell'infrastruttura e sensori interni al veicolo
- Positioning Timing (POTI): questo modulo fornisce informazioni sulle coordinate GPS della vettura (latitudine, longitudine ed altitudine). Gestisce inoltre informazioni inerenti il tempo di sistema, mantenendolo sincronizzato grazie alle informazioni ricevute ad esempio da satelliti di carreggiata
- Human Machine Interface (HMI): tramite quadro di bordo fornisce al conducente la possibilità di visualizzare una serie di informazioni come segnalazione di rischio collisione, veicolo stazionario, stato del traffico, limite di velocità, ecc
- Vehicle Data Provider (VDP): gestisce lettura e scrittura da e per rete veicolo, ovvero gestisce i dati inerenti la dinamica della vettura
- MAP Data (MAP): è un messaggio contenente le informazioni topografiche riguardanti la strada. Fornisce la mappa delle intersezioni stradali, descrivendo interi tratti o porzioni
- Signal Phase and Time (SPAT): a fronte della topografia stradale proveniente dal MAP, fornisce informazioni sullo stato attuale e il tempo di switch di un

semaforo nelle vicinanze. Le applicazioni hanno accesso a queste informazioni in quanto sono salvate nell'LDM

I moduli definiti nello standard ETSI sono:

- Decentralized Environmental Notification Message (DENM): un messaggio DENM avvisa gli utenti su una situazione eccezionale. Contiene informazioni su una condizione di pericolo stradale o di una situazione di traffico anomala, specificando il tipo di condizione (ad esempio caduta massi o ghiaccio sul manto stradale) e il luogo dove si è verificata
- Cooperative Aware Message (CAM): un messaggio CAM contiene informazioni su stato ed attributi del sistema ITS che genera il messaggio stesso, il cui contenuto varia in base al tipo di sistema che lo genera. Nel caso di veicoli le informazioni di stato contengono tempo di sistema, posizione, dinamica, tipo di veicolo ecc. Il sistema ITS che riceve un messaggio CAM diventa consapevole della presenza del sistema che origina il messaggio, della tipologia di sistema e del suo stato. Tali informazioni possono essere utilizzate in varie applicazioni ITS. Ad esempio un sistema ricevente può stimare il rischio di collisione con il sistema che origina un messaggio CAM comparando lo stato di quest'ultimo con il suo stato e se necessario avvisa il conducente tramite HMI.

I moduli definiti nello standard WAVE sono:

- Basic Safety Message (BSM): può ricevere dati dalla vettura ed inviarli al di fuori di essa. È un messaggio che viene inviato con cadenza periodica contenente informazioni come posizione, velocità, dinamica del veicolo, informazioni relative alla sicurezza, ecc
- Traveler Information Message (TIM): fornisce ai conducenti informazioni sul traffico e sulla segnaletica stradale. Ad esempio informa il conducente della presenza di un segnale "limite di velocità"
- WAVE Service Advertisment (WSA): è un messaggio periodico trasmesso da una stazione ITS per annunciarsi agli altri nodi circostanti

4.2.4 Applications/Use cases

Le informazioni ed i messaggi del layer facilities vengono utilizzate nei vari scenari applicativi o use case definiti in questo livello. Alcuni esempi reali di use case sono: frenata d'emergenza, rischio collisione frontale ed avviso di veicolo stazionario. In generale le applicazioni possono essere raggruppate in tre categorie:

- safety: comprende gli scenari applicativi che possono migliorare la sicurezza stradale, come segnalazione frenata brusca, veicolo stazionario in prossimità di un incrocio, ecc
- traffic efficiency: rientrano in questa categoria le applicazioni che hanno l'obiettivo di migliorare la gestione del traffico stradale; ad esempio introducendo stazioni ITS che gestiscono i semafori stradali in modo da ridurre il tempo di attesa ad un incrocio
- other: le applicazioni che fanno parte di questa categoria si occupano di fornire informazioni aggiuntive agli utenti, come informazioni turistiche o di servizi presenti nelle vicinanze (ospedali, parcheggi, ecc).

4.2.5 Sviluppi

In figura 4.3 sono presenti due ulteriori blocchi: MANAGEMENT e SECURITY. Entrambi saranno inseriti nel framework e copriranno aspetti inerenti ogni layer. Ad esempio gli aspetti di SECURITY saranno coperti nel layer hardware dal componente HSM; all'interno del layer facilities i messaggi BSM si potrebbero considerare attendibili in base al valori di alcuni campi; negli use case si potrebbero volere solo messaggi provenienti da un nodo con un certo ID.

Come detto in precedenza la tecnologia V2X si fonda su 802.11p. Nel giugno 2017 il gruppo 3rd Generation Partnership Project (3GPP)[9] ha rilasciato lo standard 3GPP Release 14[27], che offre migliorie al protocollo 802.11p in termini di: copertura, mobilità, minor ritardo nella comunicazione, affidabilità e scalabilità. Basandosi su questo standard 3GPP ha sviluppato la tecnologia Cellular V2X (C-V2X), che può operare in due modalità:

- Device-to-Device (D2D): comunicazioni dirette V2V, V2I e V2P basate su 802.11p
- Device-to-Network (D2N): copre il caso di comunicazioni Vehicle-to-Network (V2N) basate sulla tradizionale comunicazione cellulare, la quale abilita servizi cloud-based. In questo contesto si potrebbe costruire un'infrastruttura distribuita in cui i server V2X comunicano con le varie stazioni ITS in broadcast. Questa modalità di comunicazione offre una copertura spaziale maggiore rispetto alla classica comunicazione DSRC.

C-V2X utilizzando anche reti cellulari offre la possibilità di un futuro utilizzo della tecnologia 5G nel campo delle comunicazioni interveicolari, questo garantirebbe prestazioni ancora migliori.

Alla luce di queste considerazioni il gruppo Magneti Marelli ha deciso di rivisitare il proprio framework, come mostrato in figura 4.4.

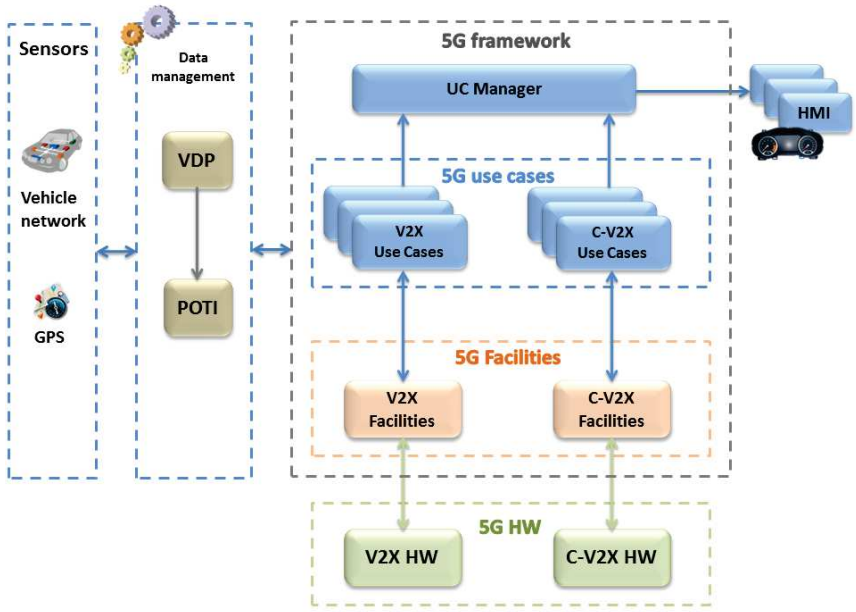


Figura 4.4: Schema a blocchi Framework Magneti Marelli

I vari elementi mostrati in figura si possono raggruppare nelle seguenti categorie: hardware, framework, gestore dati e sensori. Per quanto riguarda i livelli hardware e framwork in genere si separa il contesto C-V2X dal contesto V2X. Il caso 5G, come si vede, può essere invece considerato l'unione tra C-V2X e V2X.

L'hardware da adottare per il C-V2X è differente dall'hardware per il solo V2X dato che la prima tecnologia comprende anche le comunicazioni cellulari.

All'interno del blocco *framework* sono incluse le *facilities*, gli *use cases* e lo *use case manager*. Il livello *facilities* è stato esteso, dato che al suo interno sono adesso contenuti lo stack V2X, il middleware (compreso il S&R) e le varie facilities mostrate in figura 4.3, con l'eccezione dei moduli VDP e POTI. Anche in questo caso le facilities devono essere distinte tra C-V2X e V2X, in quanto si interfacciano con hardware differenti. Questa diversificazione come si vede si riflette anche per gli *use cases*, mentre al di sopra di questi ultimi si ha lo *use case manager*. Esso raccoglie le varie informazioni dagli use cases e le mostra nel quadro di bordo

(HMI), quindi funge da interfaccia tra questi due elementi. L'insieme di tutti i componenti visti, ad esclusione del quadro di bordo, formano il framework 5G.

Come detto in precedenza i moduli VDP e POTI non vengono più considerati all'interno del blocco facilities, infatti vengono considerati parte del blocco *data management*. Questa variazione si basa sul fatto che sia VDP che POTI gestiscono informazioni indipendenti dalla tecnologia, in quanto trattano dati della vettura (coordinate GPS e gestione del tempo di sistema) e lettura/scrittura da e per rete veicolo. Di fatti il blocco data mangement comunica con il blocco *sensors*, che ingloba la rete veicolare e sensori come GPS.

4.3 Layer Yocto

Una volta scelto l'hardware da adoperare e definita la struttura del Framework, si deve trovare un modo per fornire le ricette relative a tali informazioni al tool Yocto. Questo viene fatto tramite layer, che come si è visto, permettono di raggruppare le ricette. Sono quindi stati integrati un layer per il BSP ed uno per il Framework Magneti Marelli.

4.3.1 BSP layer

Un layer BSP contiene le ricette a supporto di uno specifico target. Avere un layer BSP custom per una particolare board significa includere ricette per tre specifiche aree: bootloader, kernel e device tree.

Per le board Udoo esiste il layer meta-udoo, mantenuto nel repository `https://github.com/graugans/meta-udoo`, che ha la struttura mostrata in figura 4.5. Il supporto alla board Udoo Quad è fornito attraverso un file di configurazione contenuto all'interno della cartella conf. All'interno del layer meta-udoo è inoltre definita la ricetta della distribuzione cross-compilata per la Udoo Quad, ovvero la udoo-image-qt5, che abilita le Qt per la distribuzione. Qt è un framework a supporto dello sviluppo di applicazioni ed interfacce utente in ambienti multipiattaforma, ovvero: desktop, embedded e mobile. Il layer meta-udoo contiene inoltre le ricette inerenti il bootloader, il kernel, il firmware e permette di generare il device tree per la board desiderata.

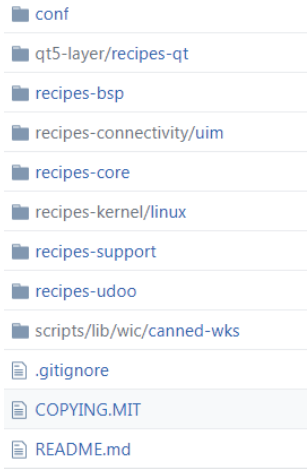


Figura 4.5: BSP Layer

Innanzitutto essendo questo layer distribuito in rete è stato inserito un file di licenza, ovvero COPYING.MIT, quindi il layer è rilasciato sotto licenza Open Source.

Come detto la Udoo Quad è provvista di un processore NXP i.MX6 ARM Cortex A9, quindi le caratteristiche di tale processore sono incluse nel file di configurazione della macchina udooqdl.conf; come mostrato nella porzione di listato di figura 4.6 sono inclusi i file:

- imx-base.inc, contenuto in un terzo layer, ovvero meta-fls-arm, fornisce impostazioni di default per i processori i.MX
- tune-cortexa9.inc, contenuto in oe-core (layer a sua volta contenuto in *Poky*) abilita specifiche ottimizzazioni per processori ARM Cortex A9
- udoo.imx-base, contiene impostazioni particolari per le board Udoo basate su questi processori

```
MACHINEOVERRIDES =. "mx6:mx6dl:mx6q:"

include conf/machine/include/imx-base.inc
include conf/machine/include/tune-cortexa9.inc
include conf/machine/include/udoo-imx-base.inc
```

Figura 4.6: Configurazioni macchina

All'interno del file udooqdl.conf sono inserite altre configurazioni molto importanti da tenere in considerazione durante il processo di build, ovvero quelle relative al kernel e al device tree, figura 4.7.

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-udooboard"
PREFERRED_VERSION_linux-udooboard ?= "3.14%"
KERNEL_IMAGETYPE = "zImage"

KERNEL_DEVICETREE ?= "imx6q-udoo-hdmi.dtb \
                      imx6q-udoo-lvds7.dtb \
                      imx6q-udoo-lvds15.dtb \
                      imx6dl-udoo-hdmi.dtb \
                      imx6dl-udoo-lvds7.dtb \
                      imx6dl-udoo-lvds15.dtb \
"
```

Figura 4.7: Configurazioni kernel

Il kernel è la versione 3.14 del kernel Linux customizzata per la board Udoo; tutte le informazioni relative al kernel si trovano all'interno del direttorio recipes-kernel/linux: ricette, impostazioni di default, patch ecc. All'interno di questo direttorio si ha un file defoconfig definito per ogni board della famiglia Udoo. In questo file sono contenute tutte le definizioni di:

- configurazioni del kernel
- setup generici
- sottosistema degli interrupt
- timer
- device tree
- ecc

In particolare il device tree potrebbe essere scritto manualmenete dallo sviluppatore nel caso in cui voglia abilitare una specifica funzione, il che richiede una grande conoscenza dell'hardware. In alternativa si può utilizzare Yocto per generarne uno automaticamente. Nel caso in esame, durante la build di una distribuzione, Yocto genera diversi device tree inerenti al tipo di interfaccia utilizzata per collegare un display alla board (se presente): HDMI, LVDS7 (schermo da 7 pollici) o LVDS15 (schermo da 15 pollici).

All'interno della directory recipes-bsp sono invece contenute un insieme di patch da applicare al bootloader U-boot in modo tale da customizzarlo per le board della famiglia Udoo.

Infine all'interno del layer sono anche fornite due particolari ricette inerenti due distribuzioni differenti:

- udoo-image-full-cmdline, fornisce un sistema operativo utilizzabile da linea di comando con il supporto ad un server SSH
- udoo-image-qt5, fornisce un sistema operativo in cui è abilitato il supporto alle qt5

4.3.2 Magneti Marelli connectivity layer

Per integrare in Yocto il Framework Magneti Marelli si è introdotto il layer, meta-mm-connectivity. Esso riflette la struttura del Framework mostrata in figura 4.4. Al suo interno sono inserite le ricette dei moduli del Framework e ricette generiche per librerie ed utility. In figura 4.8 è mostrata una porzione della struttura del layer.

La directory conf è relativa alle configurazioni dell'intero layer. Il file layer.conf contenuto all'interno di questa directory informa BitBake dell'esistenza di tutte le ricette e file append, inoltre imposta configurazioni sulla versione del layer e la priorità delle ricette.

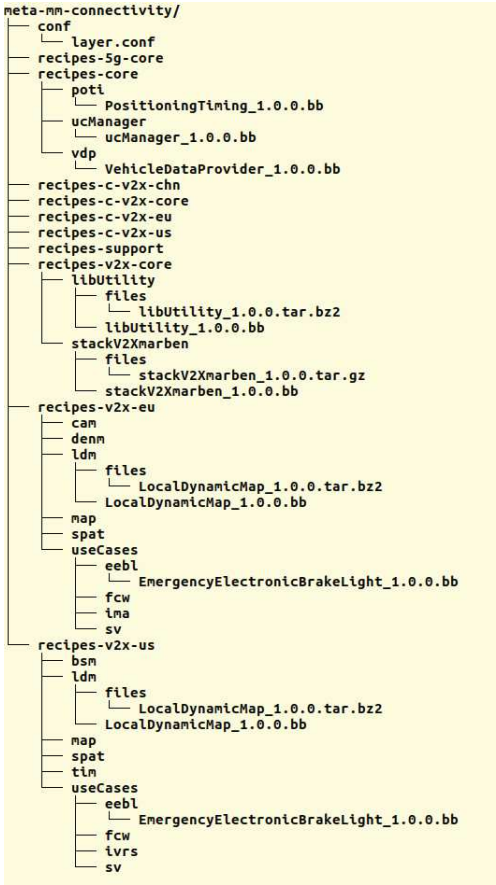


Figura 4.8: Layer MM-connectivity

All'interno delle rimanenti cartelle sono invece suddivise tutte le ricette dei vari moduli del Framework. Questo sistema di sottocartelle è stato organizzato considerando in prima analisi le varie tecnologia standard di comunicazione interveicolare, ovvero: V2X, C-V2X e 5G. Per ogni tecnologia sono stati definiti moduli indipendenti dalla tecnologia stessa; essi sono stati inseriti rispettivamente nelle cartelle: *recipes-v2x-core*, *recipes-c-v2x-core* e *recipes-5g-core*. In seconda analisi per le tecnologie V2X e C-V2X è stato deciso di introdurre un secondo livello di separazione basato sullo specifico standard (WAVE, ETSI). Per la tecnologia C-V2X è stato inoltre introdotto un livello inerente la futura standardizzazione cinese. Il layer recipes-support, momentaneamente vuoto, è stato pensato per contenere le ricette a supporto di tutto il mondo connectivity. Infine all'interno del layer recipes-core sono inseriti i moduli indipendenti alla particolare tecnologia e standard, ovvero: POTI, VDP e lo *UseCase manager*.

Analizzando in particolare le directory relative alla teconologia V2X, si distinguono:

- recipes-v2x-core: contiene le ricette comuni alla teconologia V2X. In figura sono rappresentate: le libUtility e stackV2XMarben. Le libUtility costituiscono una libreria a supporto del Framework come un sistema di log, il gestore della coda di messaggi scambiati tra i vari componenti ecc. La directory stackV2XMarben rappresenta la ricetta del blocco 802.11p Protocol Stack di

figura 4.3, in questo particolare caso si tratta del driver stack V2X fornito dalla compagnia Marben.

- recipes-v2x-eu: contiene le ricette relative allo standard ETSI
- recipes-v2x-us: contiene le ricette relative allo standard WAVE.

Questi ultimi due layer hanno strutture simili, come mostrato in figura 4.8 contengono alcuni moduli che compongono le facilities e i vari use case. Gli use case mostrati in figura sono:

- Emergency Electronic Brake Light (EEBL), rappresenta la frenata brusca
- Forward Collision Warning (FCW), rischio di collisione frontale
- Stationary Vehilcle (SV), veicolo stazionario nelle vicinanze
- In-Vehicle Road Speed Limit(IVRS), mesasggio dello standard WAVE indicante il limite di velocità
- Intresection Movement Assist (IMA), messaggio dello standard ETSI indicante la presenza di un veicolo ad un incrocio.

Infine come mostrato in figura, all'interno della directory che rappresenta il modulo LDM, oltre alla ricetta LocalDynamicMap.bb, è presente un'ulteriore directory, ovvero files. Al suo interno è stato inserito un file archivio contenente un intero progetto gestito tramite *cmake*, di fatti sono presenti: sorgenti, librerie, file header ecc. Trattandosi di un progetto CMake contiene inoltre un file CMakeList.txt, infatti per compilare il modulo all'interno della ricetta vengono ereditate le funzionalità di cmake. Questa procedura può essere utilizzata per ogni ricetta da cross-compilare.

Capitolo 5

Manuale utente

Questo capitolo è un manuale utente in cui si affrontata l'installazione, configurazione ed utilizzo del progetto Yocto.

Nella prima sezione sarà affrontata l'installazione e configurazione del sistema build su una macchina virtuale con 4 GB di memoria RAM, processore Intel i5 e sistema operativo Ubuntu 17.04.

Nella seconda sezione verrà mostrata una possibile configurazione di Toaster in modalità servizio condiviso e come configurare il sistema build per avviarlo automaticamente.

Nella terza sezione verrà mostrata la procedura di integrazione di Yocto all'interno dell'IDE Eclipse.

Nell'ultima sezione si discuterà come condurre un primo test utilizzando BitBake da terminale il cui obiettivo è generare una distribuzione per la Udoo Quad. Il processo è stato monitorato utilizzando Toaster in configurazione istanza locale al fine di raccogliere statistiche ed informazioni sul processo stesso. In seguito è mostrata la procedura per integrare il framework all'interno della distribuzione, dopo di ciò si effettua nuovamente la cross-compilazione della distribuzione.

5.1 Configurazione sistema build

In questa sezione si descrive la configurazione del sistema build in modo che possa ospitare Yocto customizzato per le board Udoo.

Innanzitutto si devono installare una serie di package, ovvero:

```
$ sudo apt-get install gawk wget git git-core diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat libsdl1.2-dev xterm sed cvs subversion coreutils \
texi2html docbook-utils python-pysqlite2 help2man make gcc g++ desktop-file-utils \
libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf automake groff curl lzop \
asciidoc u-boot-tools phablet-tools gnutls-bin picocom ncurses-dev screen virtualenv
$ sudo apt-get install gcc-arm-linux-gnueabihf
$ sudo apt-get install device-tree-compiler
```

In particolare il package virtualenv serve per creare un ambiente di lavoro virtuale che offre il vantaggio di separare i moduli eseguibili Python necessari al funzionamento di Toaster da quelli attualmente installati nel del sistema operativo della macchina build. Questa separazione permette di evitare qualunque conflitto tra versioni degli eseguibili stessi, inoltre permette di utilizzare versioni differenti di Python. Creare un ambiente virtuale non è necessario, ma è una pratica altamente raccomandata.

A questo punto si scarica il progetto da un repository tramite l'utility repo[1]. Tale repository viene clonato all'interno della *Working Directory*, dentro la quale si effettua il clone di altri due repository, cioè:

- meta-qt5: è un framework per lo sviluppo multipiattaforma di applicazioni con interfaccia grafica
- meta-intel-io-middleware che contiene l'applicazione mosquitto, ovvero un broker che implementa il protocollo di trasporto messaggi di tipo publish/subscribe MQTT.

Quanto detto si ottiene eseguendo i comandi:

```
$ mkdir ~/bin
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ PATH=${PATH}:~/bin
$ mkdir udoo-bsp
$ cd udoo-bsp
$ repo init -u https://github.com/graugans/fsl-community-bsp-platform -b krogoth
```

[1]Repo è un script Python capace di gestire un insieme di repository Git.

```
$ repo sync

$ cd ./sources/

$ git clone -b krogoth https://github.com/meta-qt5/meta-qt5.git

$ git clone -b master git://git.yoctoproject.org/meta-intel-iot-middleware
```

dai quali si evince che:

- udoo-bsp è la *Working Directory*, che in una sotto directory, sources, contiene il progetto Yocto ed i layer meta-qt5 e meta-intel-iot-middleware (per mosquitto)
- si è utilizzata la release Yocto nota come *krogoth* (release 2.1).

Una volta ottenuto Yocto, si inizializza e si attiva l'ambiente virtuale Python tramite i comandi

```
$ virtualenv venv

$ source ./venv/bin/activate
```

con i quale si crea una cartella, venv, contenente vari script Python ed una copia della libreria pip[2]. Uno di questi script è utilizzato per attivare l'ambiente virtuale, all'interno del quale si configurano le variabili d'ambiente tramite l'esecuzione dello script oe-init-build-env, che permette inoltre al sistema build di eseguire BitBake. Esso in effetti non viene invocato direttamente, ma, come si vede dal comando successivo, tramite lo script setup-environment:

```
$ MACHINE=udooqdl source ./setup-environment build
```

che crea anche la *Build Directory* (build) e due file di configurazione:

- local.conf, valorizza molte delle variabili che definiscono l'ambiente di build. Alcune delle configurazioni riguardano: la selezione del target hardware, la cartella in cui viene effettuato il fetch del codice sorgente, la cartella usata come output del processo di build, ecc. Ad esempio la macro MACHINE??=udooqdl indica il target hardware per il quale si sta effettuando la build. La macro DISTRO?=poky è assegnata di default ed identifica la distribuzione da generare
- bblayer.conf, indica a BitBake i layer da considerare durante la build. Di default si ha una lista di layer richiesti, tuttavia nel caso di layer customizzati questi devono essere aggiunti manualmente.

[2]pip è un sistema di gestione pacchetti usato per installare e gestire package Python.

Subito dopo sono stati modificati alcuni file per ottenere l'effetto desiderato. Il primo file da modificare è bblayer.conf, nel quale si aggiungono ulteriori layer, come mostrato nel file A.1.1.

La seconda modifica riguarda applicazioni e versioni da integrare all'interno della distribuzione, modificando direttamente local.conf, come mostrato nel file A.1.2. In questo modo le modifiche sarebbero applicate a qualunque distribuzione generata. In alternativa si possono modificare i file:

- tcmode-default.inc, A.1.3, contenuto in sources/poky/meta/conf/distro/include, che rappresenta il file di configurazione della distribuzione al quale aggiungere le modifiche inerenti le versioni dei package da integrare. Questo significa che si devono includere all'interno dei layer anche le ricette relative alle versioni desiderate
- udoo-image-qt5, A.1.4, rappresenta la ricetta dell'immagine, è contenuto all'interno del direttorio sources/meta-udoo/qt5-layer/recipes-qt/images. In questo file vanno aggiunte le applicazioni da installare all'interno dell'immagine.

Un'ulteriore considerazione riguarda il compilatore GCC versione 4.8, in particolare è stato riscontrato un errore in fase di build dovuto al compilatore nativo GCC versione 6. Il problema è stato risolto tramite la patch A.1.5 che deve essere inclusa nel file gcc-4.8.inc inserendo al suo interno il nome della patch:

```
file://0076-Fix-build-with-gcc-6.patch
```

Le configurazioni del sistema build effettuate in questa sezione sono usate come punto di partenza per il test eseguito in sezione 5.4.

5.2 Toaster

Toaster come detto offre un ambiente GUI al tool BitBake, fornisce un web server e un database rispettivamente per effettuare richieste tramite l'interfaccia web e collezionare informazioni da utilizzare per scopi di analisi. Nel caso di istanza locale tutti questi strumenti sono raggruppati all'interno di un unico host, mentre nel caso di servizio condiviso si utilizza un'architettura distribuita. Quest'ultimo contesto permette a diversi utenti di utilizzare server build per gestire interi progetti e cross-compilare ricette da remoto.

In questa sezione si mostra come configurare all'interno di un unico ambiente un servizio condiviso.

Innanzitutto si effettuano le configurazioni del sistema host viste nella sezione 5.1. L'unica differenza consiste nell'impostazione della *Working Directory*, invece di

utilizzare la directory udoo-bsp viene creata ed utilizzata una directory Ubuntu destinata a contenere file che rappresentano informazioni dinamiche, ovvero /var/www/toaster. Toaster viene configurato per utilizzare il web server Apache e il DataBase Management System (DBMS) MySQL[25], per cui si installano i seguenti package:

```
$ sudo apt-get install apache2 libapache2-mod-wsgi mysql-server \
virtualenv libmysqlclient-dev
```

Il package libapache2-mod-wsgi è un modulo Apache che permette al server web di interfacciarsi con applicazioni Django (Toaster è un applicativo scritto in Django) tramite Web Server Gateway Interface (WSGI)[3].

MySQL è invece utilizzato come alternativa al database di default, ovvero SQLite, in quanto offre il vantaggio di essere altamente personalizzabile e pone particolare attenzione alla sicurezza, ad esempio assegna privilegi come creazione di un database o cancellazione di dati a specifici utenti o gruppi di utenti.

Toaster per essere eseguito richiede inoltre che siano soddisfatte ulteriori dipendenze Python, nel senso che richiede l'installazione delle applicazioni specificate nel file toaster-requirements.txt. Prima di installare tramite *pip* tali applicazioni deve essere configurato ed inizializzato l'ambiente virtuale tramite virtualenv:

```
$ cd /var/www/toaster/
$ virtualenv venv
$ source ./venv/bin/activate
$ (venv) pip install -r ./sources/poky/bitbake/toaster-requirements.txt
$ (venv) pip install mysql
```

L'attivazione dell'ambiente virtuale all'interno del quale sarà eseguito Toaster dove essere fatta tutte le volte che si avvia il servizio stesso.

A questo punto si creano database ed account utente MySQL al quale vengono forniti tutti i privilegi di gestione e modifica sul database stesso. Come detto questi servono a Toaster per collezionare informazioni sull'esecuzione del processo di cross-compilazione:

```
$ mysql -u root -p
mysql> CREATE DATABASE toaster_data;
mysql> CREATE USER 'toaster'@'localhost' identified by 'password';
```

[3]WSGI è un protocollo di trasmissione che descrive come avviene la comunicazione tra l'applicazione e il server web

```
mysql> GRANT all on toaster_data.* to 'toaster'@'localhost';

mysql> quit
```

Ogni applicazione web basata sul framework Django contiene un file settings.py che ne definisce le impostazioni. Di seguito sono mostrate le modifiche a questo file:

```
DATABASES = {

  'default': {

    'ENGINE': 'django.db.backends.mysql',

    'NAME': 'toaster_data',

    'USER': 'toaster',

    'PASSWORD': 'password',

    'HOST': 'localhost',

    'PORT': '3306',

  }

SECRET_KEY = 'secretKey'

STATIC_ROOT = '/var/www/toaster/static_files/'
```

In DATABASES sono contenute tutte le impostazioni sul database, che rispecchiano le impostazione del database MySQL precedentemente creato. SECRET_KEY è il valore della chiave segreta usata per cifrare i dati, deve essere unica e impredicibile. STATIC_ROOT è il path assoluto all'interno del quale vengono conservati i file statici che servono all'applicazione in fase di sviluppo.

Un secondo file importante all'interno di applicazioni Django è il file manage.py che definisce un set di comandi utilizzabili dal programmatore per la gestione e il mantenimento dell'applicazione stessa. Nel caso di Toaster si eseguono i seguenti comandi:

```
$ cd /var/www/toaster/sources/poky/

$ ./bitbake/lib/toaster/manage.py syncdb

$ ./bitbake/lib/toaster/manage.py migrate

$ TOASTER_DIR=‘pwd‘ TOASTER_CONF=./meta-poky/conf/toasterconf.json \

./bitbake/lib/toaster/manage.py checksettings

$ ./bitbake/lib/toaster/manage.py collectstatic
```

In particolare syncdb e migrate creano lo *schema* del database. Il comando checksetting permette di verificare se le impostazioni esistenti sono valide o meno, infine collectstatic colleziona i file statici nella directory puntata da STATIC_ROOT.

Dopo la sezione di impostazioni del database, viene creato un file di configurazione Apache per l'applicazione Toaster, il cui percorso assoluto è /etc/apache2/conf-available/toaster.conf (A.2.1). Al suo interno viene specificato la directory contenente i file statici dell'applicazione stessa, la versione della libreria Python utilizzata per avviare l'applicazione e lo script che definisce le configurazioni del protocollo WSGI per Toaster. A questo punto si abilità il modulo WSGI e si applicano le configurazioni contenute nel file A.2.1; per rendere effettive le modifiche infine si riavvia Apache:

```
$ sudo a2enmod wsgi

$ sudo service apache2 restart

$ sudo a2enconf toaster

$ chmod +x bitbake/lib/toaster/toastermain/wsgi.py

$ sudo service apache2 restart
```

Lo step successivo consiste nel creare una procedura di avvio automatico di Toaster all'avvio del sistema build. Innanzitutto si crea lo script runbuilds-service.sh (A.2.2), contenuto nella directory /var/www/toaster/sources/poky/bitbake/lib/toaster/, al quale devono essere forniti i permessi di esecuzione. Esso si occupa di avviare Toaster tramite l'esecuzione del binario bitbake/bin/toaster come si vede nell'ultima riga del file A.2.2. Questo script viene invocato all'interno di un servizio *systemd*[4] preparato appositamente. Per configurare il servizio è stato scritto un ulteriore file di configurazione, runbuilds.conf, contenuto nella cartella /etc/systemd/system. Come si vede all'interno del file A.2.3 l'avvio dello script runbuilds-service viene effettuato passandogli il parametro start, che viene poi inoltrato al binario bitbake/bin/toaster. In questo modo Toaster viene configurato per mettersi in attesa di richieste all'indirizzo di default, ovvero 127.0.0.1:8000.

Infine si manda in esecuzione il servizio tramite i seguenti comandi:

```
# service runbuilds start

$ sudo su - mm-develop

$ screen -rS toaster
```

All'interno del file di configurazione del servizio A.2.3 viene avviata anche l'applicazione screen, un gestore di finestre che permette di condividere la stessa sessione di terminale tra più processi. Essa viene avviata in modalità *detached*, ovvero la sessione viene scollegata dal terminale attuale e posta in una finestra di background;

[4]systemd è un gestore di sistema e di servizi utilizzato in ambiente Linux, può essere utilizzato per avviare automaticamente all'avvio del sistema qualunque servizio

per riesumarla dallo stato detached si usa l'ultimo comando del listato precedente. A questo punto ci si può collegare con un qualunque browser all'indirizzo 127.0.0.1:8000 che mostra la pagina iniziale di Toaster, figura 5.1.

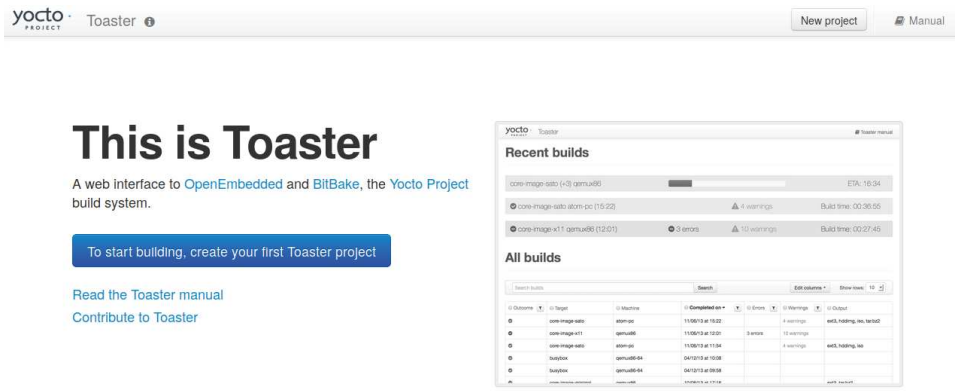


Figura 5.1: Pagina Iniziale Toaster

5.3 Eclipse plug-in

Il plug-in per l'IDE Eclipse permette di utilizzare gli strumenti forniti dalla SDK all'interno dell'IDE stesso. In questa sezione si mostra come integrare Yocto all'interno di Eclipse, in particolare è stata usata la versione Luna SR2 (4.4.2) scaricabile da `http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/lunasr2`. Oltre all'IDE è necessario installare una serie di estensioni aggiuntive. Dal menù *Help* selezionare *Install New Software* e successivamente *Luna - http://download.eclipse.org/releases/luna* nella sezione *Work with*. Nel box sottostante sono presenti un elenco di pacchetti da installare divisi per sezione, quindi selezionare i seguenti:

- Linux Tools
  - Linux Tools LTTng Tracer Control
  - Linux Tools LTTng Userspace Analysis
  - LTTng Kernel Analysis
- Mobile and Device Development

  - C/C++ Remote Launch (Requires RSE Remote System Explorer)
  - Remote System Explorer End-user Runtime
  - Remote System Explorer User Actions
  - Target Management Terminal (Core SDK)
  - TCF Remote System Explorer add-in
  - TCF Target Explorer
- Programming Languages
  - C/C++ Autotools Support
  - C/C++ Development Tools

Un ulteriore plug-in da integrare all'IDE è Eclipse Yocto plug-in. Nuovamente dal menù *Help* selezionare *Install New Software* ed inserire l'URL `http://downloads.yoctoproject.org/releases/eclipse-plugin/2.1.1/luna` ed un nome significativo per tale URL cliccando su *Add* nell'area *Work with*. In seguito da quest'ultima selezionare l'URL appena inserita ed installare:

- Yocto Project ADT Plug-in
- Yocto Project Bitbake Commander Plug-in
- Yocto Project Documentation plug-in

A questo punto tutto ciò che è necessario al funzionamento del plug-in è installato.

Ad esempio si potrebbe creare un nuovo progetto *Yocto Project Bitbake Commander* dall'interfaccia dell'IDE, valorizzando i campi *Project Name* e *Project Location* in modo appropriato; ovvero deve essere fornito il path contenente una versione valida del progetto Poky. In questo caso considerando il path in cui è stato clonato il repository nella sezione 5.1, i due campi sopra citati avranno rispettivamente i valori poky e /your-home-directory/udoo-bsp/sources. Al progetto appena creato si possono successivamente apportare tutte le modifiche desiderete: aggiungere o modificare ricette e creare nuovi layer.

All'interno dell'IDE sono fornite procedure guidate che permettono di creare layer BSP e nuove ricette. Tali procedure creano tutto il necessario per avere rispettivamente layer e ricetta conformi al progetto Yocto.

In figura 5.2 è mostrato il risultato della creazione di un layer BSP da Eclipse, il quale si basa sullo script yocto-bsp fornito all'interno di *Poky*. Come si vede viene creata tutta la struttura del layer:

- directory conf contenente il file layer.conf e la directory machine che a sua volta contiene un file di configurazione della macchina
- ulteriori directory contenenti un set di ricette generate automaticamente

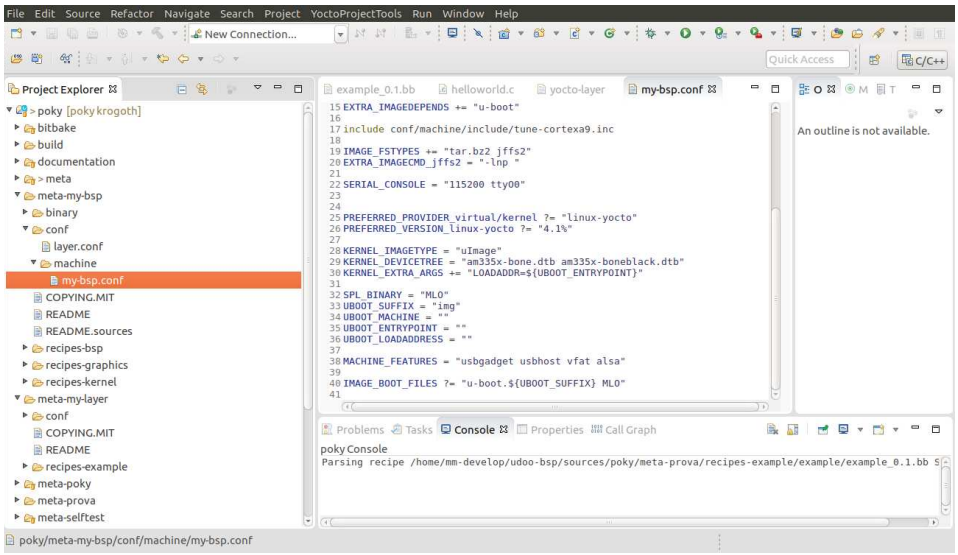


Figura 5.2: BSP layer in Eclipse

Tale struttura può facilmente essere utilizzata come punto di partenza per personalizzare il proprio BSP.

Queste sono le uniche due procedure integrate in Eclipse, ma avendo creato un progetto Yocto integrato in Eclipse, tutte le modifiche alla struttura effettuate dagli script forniti da *Poky* si riflettono nell'IDE. Ad esempio nel caso in cui si voglia crare un layer generico è possibile utilizzare lo script yocto-layer ed in seguito riaggiornare il progetto all'interno di Eclipse.

L'IDE per effettuare una build si affida infine a Toaster che può essere lanciato direttamente dall'IDE stesso selezionando *Launch Toaster* ed inserendo l'URL impostato per il suo funzionamento, in questo caso 127.0.0.1:8000.

Eclipse può anche essere utilizzato come strumento di sviluppo applicativo per il target. Prima di procedere è necessario avere la corretta toolchain ed il root filesystem del target.

La toolchain viene generata da bitbake, cross-compilando la ricetta:

```
$ MACHINE=udooqdl bitbake udoo-image-qt5 -c populate-sdk
```

che genera uno script di installazione dell'SDK. L'esecuzione di questo script crea anche una directory contenente il root filesystem del target.

In figura 5.3 si devono valorizzare rispettivamente i campi *Toolchain Root Location* e *Sysroot Location* con i corretti path della toolchain e del root filesystem del target.

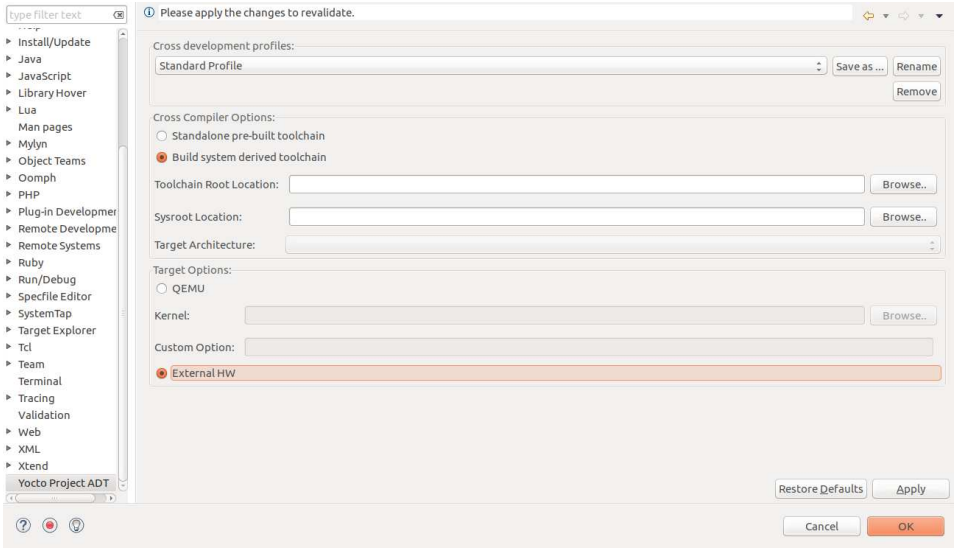


Figura 5.3: Configurazione ADT Eclipse

In questo modo Eclipse è in grado di creare e gestire progetti software per il target hardware.

5.4 Generazione distribuzione

Il test eseguito in questa sezione utilizza le configurazioni del sistema build viste in sezione 5.1. Al fine di collezionare informazioni sul lavoro svolto dal processo di build è stato avviato Toaster in modalità istanza locale. Innanzitutto si installano le dipendenze Python necessarie all'esecuzione di Toaster definite nel file toaster-requirements.txt. In seguito viene eseguito il binario che avvia Toaster:

```
$ pip install -r sources/poky/bitbake/toaster-requirements.txt
```

```
$ source ../source/poky/bitbake/bin/toaster
```

Il secondo comando deve necessariamente essere eseguito prima di iniziare il processo di build, altrimenti Toaster non sarà in grado di collezionare le informazioni

sull'esecuzione del processo stesso. Dopo aver avviato questo comando si può accedere all'interfaccia web di Toaster tramite qualunque browser all'indirizzo di default, ovvero 127.0.0.1:8000. Toaster, non appena viene avviata la build, crea un progetto con il nome di *Command Line Builds*.

Una volta configurato l'ambiente di lavoro è stata effettuata la build dell'immagine. udoo-image-qt5 tramite l'esecuzione del comando:

```
$ MACHINE=udooqdl bitbake udoo-image-qt5
```

Il sistema di build BitBake elabora la ricetta, risolvendo automaticamente tutte le dipendenze ad essa associate; ovvero elabora le ricette necessarie alla cross compilazione di questa ricetta. Questo lavoro viene compiuto in modo ricorsivo per ogni ricetta da elaborare fino alla risoluzione di tutte le dipendenze ed alla generazione della distribuzione. Tale processo è monitorabile tramite Toaster come mostrato in figura 5.4, mentre al termine del processo viene mostrata un'interfaccia simile a quella di figura 5.5.

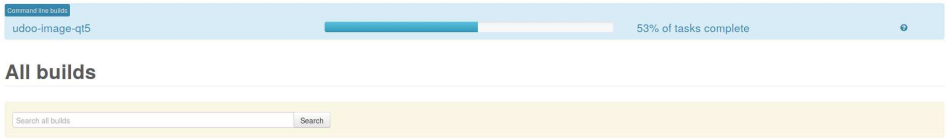


Figura 5.4: Avanzamento processo build

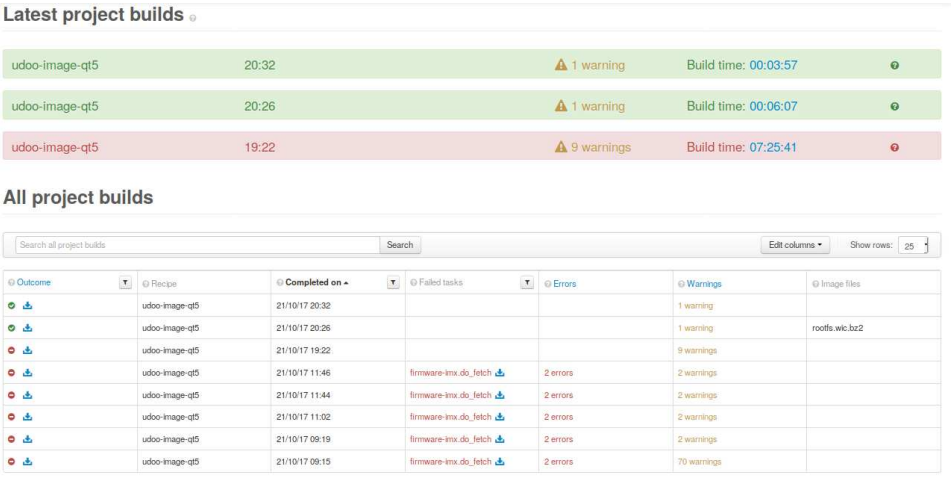


Figura 5.5: Interfaccia progetti

In particolare Toaster, dopo terminato il processo di build della ricetta, colleziona il risultato sia in caso di build realizzata con successo sia in caso di errore, come mostrato in figura 5.5. Questa prima interfaccia fornisce informazioni generali come il tempo di esecuzione della build, eventuali fallimenti e le ragioni che li hanno causati.

Per build che terminano con errori, si ha una situazione simile a quella mostrata in figura 5.6. In questi casi Toaster fornisce sia un sommario dei messaggi di errore, sia un log scaricabile dall'interfaccia stessa contenente tutti i dettagli del task, semplificando all'utente il compito di risalire alla causa che ha generato l'errore.

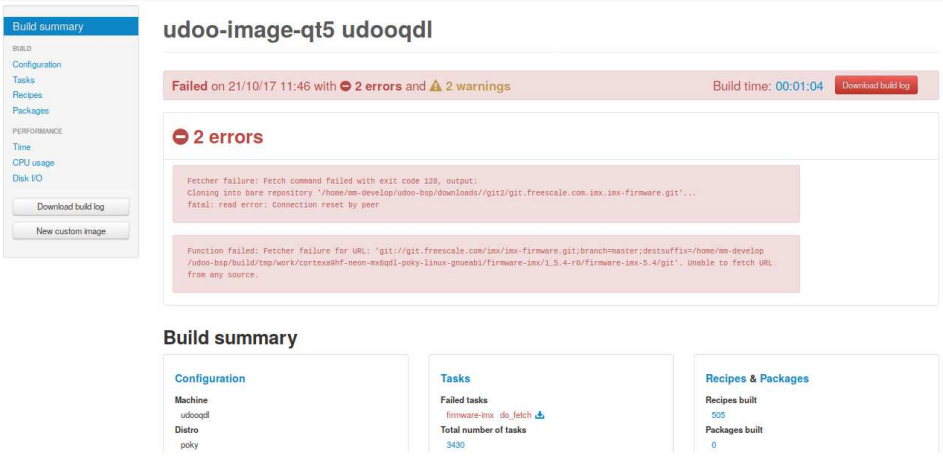


Figura 5.6: Dettaglio errore

A questo punto si procede ad integrare le ricette del Framework. Innanzitutto viene creata la directory meta-mm-connectivity con l'ausilio dell'utility yocto-layer create a cui viene dato come argomento il nome del layer privo del prefisso meta-, il quale sarà aggiunto in automatico dall'utility.

In seguito viene creata la struttura di direttori presentata nella sezione 4.3.2. Come esempio si mostra la cross-compilazione della ricetta libUtility_1.0.0 (A.1.6), in cui:

- SUMMARY: indica una descrizione della funzionalità della ricetta
- SECTION: indica la sezione nella quale il package generato sarà categorizzato. Essendo una libreria essa verrà categorizzata come libs
- LICENSE: indica il tipo di licenza della ricetta

- LIC_FILES_CHKSUM: indica il file di testo usato come licenza ed il checksum di quel file. Se quest'ultimo viene modificato si può avere un fallimento durante il processo di build in quanto se non si aggiorna il checksum non si ha la corretta corrispondenza
- DEPENDS: elenca le dipendenze della ricetta, che possono essere altre ricette. Il sistema di build deve assicurarsi di aver già cross-compilato tutte le dipendenze e che il contenuto è presente nell'appropriata locazione del root filesystem prima di poter cross-compilare la ricetta in questione
- PN: è una variabile con duplice utilizzo. Può riferirsi al nome della ricetta nel contesto di un file usato dal sistema di build OE per creare il package in uscita. Nel caso in questione il nome della ricetta è libUtility_1.0.0 per cui PN è libUtility. PN si riferisce invece al nome del package nel contesto di un file creato o generato dal sistema di build. PN può infine avere dei suffissi come -cross o -native e dei prefissi come lib64- o lib32-
- S: rappresenta la locazione della *Build Directory* all'interno del quale verrà eseguita la fase di unpack del codice sorgente da cross-compilare. Il nome del path è costruito a partire dal valore della *Working Directory* al quale viene concatenato il nome della ricetta (il valore della variabile PN privata di eventuali suffissi e prefissi) ed il numero di versione della ricetta indicato dalla variabile PV (il valore di questa variabile è estratto nuovamente dal nome della ricetta)
- SRC_URI: indica la locazione dei file sorgenti, siano essi locali (come in questo caso) o remoti
- EXTRA_OECMAKE: fornisce a cmake opzioni di compilazione aggiuntive. In questo caso si abilita la modalità *verbose* che fornisce maggiori dettagli
- TARGET_CXXLAGS: specifica le opzioni da passare al compilatore C++ durante il processo di build per il target. In questo caso cmake viene configurato per tenere in considerazione un particolare wrapper costruito su un sistema di log, MMLOGGER
- inherit cmake: *inherit* rappresenta una direttiva, usata quando si scrive una ricetta o una classe per farle ereditare le funzionalità di un'altra classe. In questo caso si dice alla ricetta di ereditare le funzionalità di cmake per effettuare il build automatico dei sorgenti.

Per integrare la ricetta all'interno della distribuzione si può seguire la stessa procura effettuata nella sezione 5.1, ovvero modificando i file:

- local.conf A.1.2, concatenando alla variabile INSTALL_IMAGE_append il nome della ricetta

- bblayer.conf A.1.1, concatenando il percorso della directory contenente alla ricetta alla variabile BBLAYERS.

Infine si può rilanciare la generazione della ricetta udoo-image-qt5, che rispetto alla prima volta effettua la procedura di build solo per la ricetta del Framework, impiegando quindi molto meno tempo. In altre parole è stata effettuata una procedura di build incrementale, in quanto alla distribuzione è stata aggiunta una feature non presente inizialmente.

Capitolo 6

Conclusioni

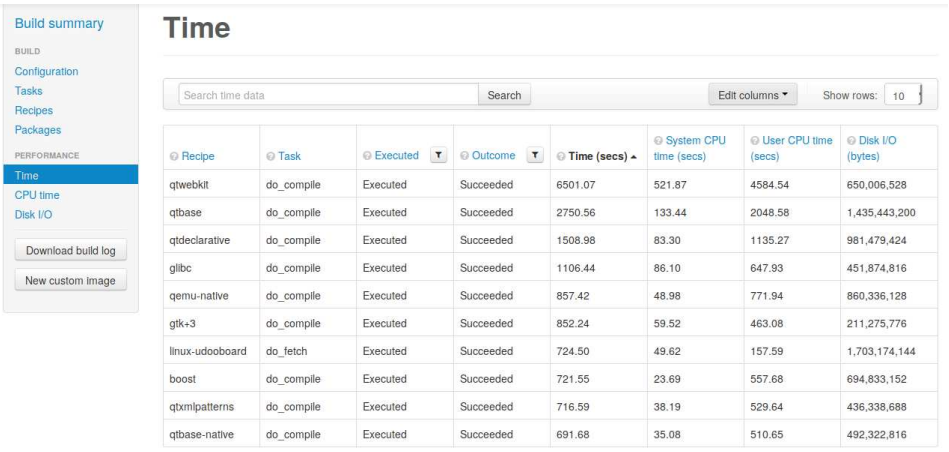
All'interno di questo capitolo sono raccolti e discussi i risultati ottenuti delle due cross-compilazioni effettuate grazie alle configurazioni viste nella sezione 5.4.

Nella seconda sezione si affrontano le problematiche affrontate e risolte durante lo svolgimento della tesi, mentre nell'ultima sezione si mostrano possibili scenari futuri inerenti nuove configurazioni ed utilizzi del progetto Yocto.

6.1 Risultati

Il tempo necessario a completare il processo di cross-compilazione della ricetta senza l'integrazione del modulo del Framework, a fronte delle caratteristiche hardware del sistema build e della velocità della connessione Internet, è stato di: 19 ore e 25 minuti. Ovviamente al migliorare di entrambi questi fattori diminuisce il tempo necessario ad ottenere la prima cross-compilazione. In ogni caso la prima cross-compilazione di una qualunque ricetta è sempre quella che impiega maggior tempo, in quanto bitbake deve svolgere tutti i compiti richiesti dalla ricetta stessa.

Dopo aver effettuato la build, come mostrato in figura 6.1, si possono verificare i dettagli statistici del processo relativi alle varie ricette ed ai vari step eseguiti da BitBake in termini di: tempo di esecuzione del task in secondi, utilizzo della CPU e volume di dati di I/O su disco rigido.



Recipe	Task	Executed	Outcome	Time (secs)	System CPU time (secs)	User CPU time (secs)	Disk I/O (bytes)
qtwebkit	do_compile	Executed	Succeeded	6501.07	521.87	4584.54	650,006,528
qtbase	do_compile	Executed	Succeeded	2750.56	133.44	2048.58	1,435,443,200
qtdeclarative	do_compile	Executed	Succeeded	1508.98	83.30	1135.27	981,479,424
glibc	do_compile	Executed	Succeeded	1106.44	86.10	647.93	451,874,816
qemu-native	do_compile	Executed	Succeeded	857.42	48.98	771.94	860,336,128
gtk+3	do_compile	Executed	Succeeded	852.24	59.52	463.08	211,275,776
linux-udooboard	do_fetch	Executed	Succeeded	724.50	49.62	157.59	1,703,174,144
boost	do_compile	Executed	Succeeded	721.55	23.69	557.68	694,833,152
qtxmlpatterns	do_compile	Executed	Succeeded	716.59	38.19	529.64	436,338,688
qtbase-native	do_compile	Executed	Succeeded	691.68	35.08	510.65	492,322,816

Figura 6.1: Statistiche tempo di esecuzione

In particolare per quanto riguarda i dati inerenti l'utilizzo di CPU si distinguono:

- *System CPU time*, indica il tempo speso dalla CPU in kernel mode
- *User CPU time*, indica il tempo speso dalla CPU in user mode

Il risultato ottenuto dalla cross-compilazione della distribuzione con integrate le ricette del framework è mostrato in figura 6.2. Rispetto al primo caso il tempo di compilazione è drasticamente diminuito, di fatti è stata completata in meno di otto minuti.

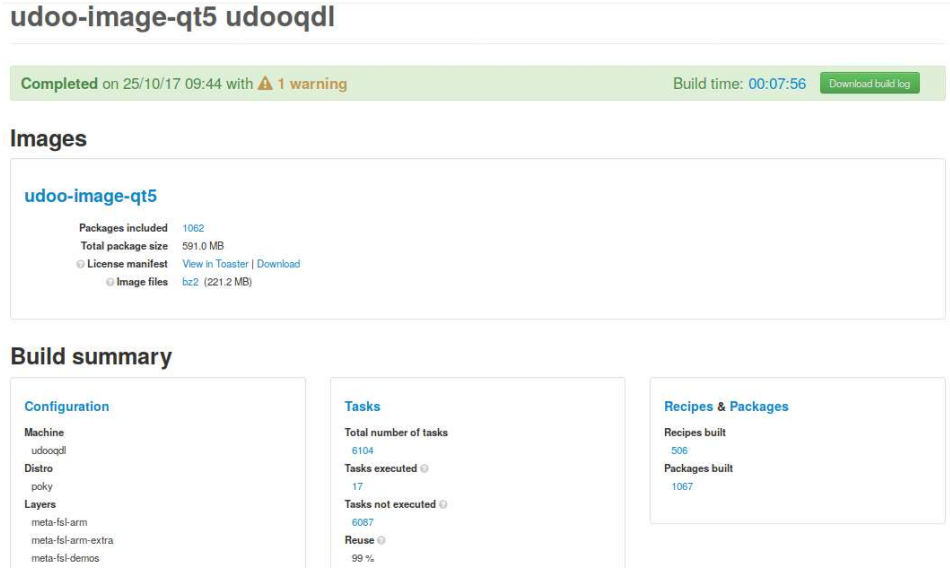


Figura 6.2: Integrazione framework

Questo è dovuto al fatto che il numero di task realmente eseguiti da bitbake è decisamente minore rispetto al numero di task totali necessari alla generazione della distribuzione. Come mostrato all'interno del box *Tasks* in figura 6.2, sono stati eseguiti solo 17 task a fronte dei 6104 task totali, per cui sono stati riutilizzati i risultati di 6087 task (quelli della prima cross-compilazione) che in percentuale rappresentano oltre il 99% del totale. In generale i tempi di cross-compilazione al fine di integrare una nuova ricetta all'attuale distribuzione sono dell'ordine dei minuti, dunque con un guadagno medio di circa il 95%. Come si vede da questi numeri, questo è un grande vantaggio di Yocto, che è valido non solo per la cross-compilazione della stessa ricetta, ma si applica anche al caso di ricette diverse. Se due ricette condividono alcuni task, questi saranno eseguiti solo per la prima cross-compilazione della prima ricetta.

6.2 Problematiche risolte

Durante questo lavoro di tesi sono state affrontate e risolte due grandi problematiche.

La prima è sorta a causa di un problema strutturale del progetto Yocto, legato al suo malfunzionamento nel caso in cui si utilizza una rete protetta da proxy. Tale problema è dovuto al fatto che Yocto fa un grosso uso di strumenti come git e wget sia per fare il fetch del progetto stesso sia per il fetch dei pacchetti effettuato all'interno delle ricette. Questi strumenti a loro volta presentano problemi con reti protette. In rete esistono varie guide e workaround che dovrebbero risolvere il problema, ma nessuna di questa ha portato al risultato sperato nel caso delle rete aziendale Magneti Marelli. Per risolvere il problema si è utilizzato un PC esterno a tale rete su cui è stato configurato un server ssh, che è stato utilizzato come vero e proprio sistema di build in quanto i comandi venivano forniti da un client ssh posto internamente alla rete aziendale.

La seconda problematica è stata trovare il corretto versioning dei vari pacchetti da installare per creare una distribuzione funzionante per la Udoo Quad. Il vero problema è stato applicare le dovute patch in modo tale che il processo di build terminasse correttamente. Ad esempio è stato riscontrato che cross-compilare GCC versione 5 (o superiore) utilizzando la GCC versione 6 nel sistema di build non comporta nessun problema. Nel caso in esame era richiesto cross-compilare GCC versione 4.8, il che ha evidenziato problemi di compatibilità tra le due versioni. Questo problema è stato risolto aggiungendo una patch appropriata. Infine è da notare che essendo Yocto un progetto opensource e che i layer utilizzati sono mantenuti da una comunità aperta di sviluppatori è possibile che tali layer verranno modificati in futuro. Ad esempio il layer meta-intel-iot-middleware è stato abbandonato e le sue ricette (come mosquitto) sono state integrate altri layer.

6.3 Sviluppi futuri

Yocto è sicuramente uno strumento molto potente se compreso ed utilizzato attentamente.

Uno dei possibili scenari futuri riguarda la configurazione di Toaster. Nella sezione 5.2 si è impostato Toaster come servizio condiviso, il quale permette di utilizzare un server come sistema build, mentre i client hanno solo l'onere di creare il progetto correttamente configurato. Durante il lavoro svolto nella tesi si è però utilizzato come indirizzo del servizio Toaster il locahost (127.0.0.1). Si potrebbe quindi pensare di creare un'infrastruttura client/server distribuita in cui su uno

o più server verrebbero distribuiti i componenti operativi (BitBake e database). Lato client si andrebbe allora a configurare Toaster in modo da contattare questi server e fargli svolgere il lavoro richiesto. Il grande vantaggio di queste genere di approccio sarebbe quello di sgravare i vari client dell'alto carico computazionale necessario ad eseguire il lavoro, inoltre tramite Toaster essi avrebbero la possibilità di tenere traccia del lavoro svolto durante il processo di build.

Un secondo aspetto di interesse sarà generare un layer BSP adatto alla board proprietaria del gruppo Magneti Marelli. Sicuramente come punto di partenza potrà essere utilizzato il layer BSP della Udoo Quad, il quale dovrà però essere modificato considerando le diverse caratteristiche hardware tra le due board.

Appendice A

Configurazione ambiente Yocto per Magneti Marelli

A.1 Configurazione host

A.1.1 File bblayer.conf

```
LCONF_VERSION = "7"
BBPATH = "${TOPDIR}"
BSPDIR := "${@os.path.abspath(os.path.dirname(d.getVar('FILE', True)) +
         '/../..')}"
BBFILES ?= ""
BBLAYERS = " \
${BSPDIR}/sources/poky/meta \
${BSPDIR}/sources/poky/meta-yocto \
${BSPDIR}/sources/meta-intel-iot-middleware \
\
${BSPDIR}/sources/meta-openembedded/meta-oe \
${BSPDIR}/sources/meta-openembedded/meta-multimedia \
${BSPDIR}/sources/meta-openembedded/meta-python \
${BSPDIR}/sources/meta-openembedded/meta-networking \
```

```
\
${BSPDIR}/sources/meta-qt5 \
${BSPDIR}/sources/meta-openembedded/meta-ruby \
\
${BSPDIR}/sources/meta-fsl-arm \
${BSPDIR}/sources/meta-fsl-arm-extra \
${BSPDIR}/sources/meta-udoo \
${BSPDIR}/sources/meta-fsl-demos \ "
```

A.1.2 File local.conf

```
MACHINE ??= 'udooqdl'
DISTRO ?= 'poky'
PACKAGE_CLASSES ?= "package_ipk"
GCCVERSION = "4.8%"
BINUVERSION = "2.26%"
GLIBCVERSION = "2.22%"
PREFERRED_VERSION_boost = "1.57.0"
EXTRA_IMAGE_FEATURES = "debug-tweaks"
IMAGE_INSTALL_append = " canutils tree dfu-util libconfig boost eglinfo-x11 \
glmark2 libsocketcan ppp-dialin ppp iptables usbutils mosquitto c-ares libxml2 \
sqlite3 valgrind openssh"
USER_CLASSES ?= "buildstats image-mklibs"
PATCHRESOLVE = "noop"
BB_DISKMON_DIRS = "\
STOPTASKS,${TMPDIR},1G,100K \
STOPTASKS,${DL_DIR},1G,100K \
STOPTASKS,${SSTATE_DIR},1G,100K \
STOPTASKS,/tmp,100M,100K \
ABORT,${TMPDIR},100M,1K \
ABORT,${DL_DIR},100M,1K \
ABORT,${SSTATE_DIR},100M,1K \
ABORT,/tmp,10M,1K"
PACKAGECONFIG_append_pn-qemu-native = " sdl"
```

```
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
CONF_VERSION = "1"
DL_DIR ?= "${BSPDIR}/downloads/"
ACCEPT_FSL_EULA = "1"
```

A.1.3 File tcmode-default.conf

```
#
# Default toolchain configuration
#
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}binutils =
                "binutils-cross-${TARGET_ARCH}"
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}gcc-initial =
                "gcc-cross-initial-${TARGET_ARCH}"
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}gcc = "gcc-cross-${TARGET_ARCH}"
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}g++ = "gcc-cross-${TARGET_ARCH}"
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}compilerlibs = "gcc-runtime"
PREFERRED_PROVIDER_gdb = "gdb"
PREFERRED_PROVIDER_virtual/${SDK_PREFIX}binutils-crosssdk ?=
                "binutils-crosssdk-${SDK_ARCH}"
PREFERRED_PROVIDER_virtual/${SDK_PREFIX}gcc-initial =
                "gcc-crosssdk-initial-${SDK_ARCH}"
PREFERRED_PROVIDER_virtual/${SDK_PREFIX}gcc = "gcc-crosssdk-${SDK_ARCH}"
PREFERRED_PROVIDER_virtual/${SDK_PREFIX}g++ = "gcc-crosssdk-${SDK_ARCH}"
PREFERRED_PROVIDER_virtual/${SDK_PREFIX}compilerlibs = "nativesdk-gcc-runtime"
# Default libc config
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}libc-for-gcc = "${TCLIBC}"
PREFERRED_PROVIDER_virtual/nativesdk-${SDK_PREFIX}libc-for-gcc ?=
                "nativesdk-glibc"
PREFERRED_PROVIDER_virtual/${TARGET_PREFIX}libc-initial = "${TCLIBC}-initial"
PREFERRED_PROVIDER_virtual/nativesdk-${SDK_PREFIX}libc-initial ?=
                "nativesdk-glibc-initial"
PREFERRED_PROVIDER_virtual/gettext ??= "gettext"
GCCVERSION ?= "4.8%"
```

```
SDKGCCVERSION ?= "${GCCVERSION}"
BINUVERSION ?= "2.26%"
GDBVERSION ?= "7.10%"
GLIBCVERSION ?= "2.22"
UCLIBCVERSION ?= "1.0%"
LINUXLIBCVERSION ?= "4.4"
PREFERRED_VERSION_gcc ?= "${GCCVERSION}"
PREFERRED_VERSION_gcc-cross-${TARGET_ARCH} ?= "${GCCVERSION}"
PREFERRED_VERSION_gcc-cross-initial-${TARGET_ARCH} ?= "${GCCVERSION}"
PREFERRED_VERSION_gcc-crosssdk-${SDK_ARCH} ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_gcc-crosssdk-initial-${SDK_ARCH} ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_gcc-cross-canadian-${TRANSLATED_TARGET_ARCH} ?= "${GCCVERSION}"
PREFERRED_VERSION_gcc-runtime ?= "${GCCVERSION}"
PREFERRED_VERSION_gcc-sanitizers ?= "${GCCVERSION}"
PREFERRED_VERSION_nativesdk-gcc-runtime ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_nativesdk-gcc-sanitizers ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_libgcc ?= "${GCCVERSION}"
PREFERRED_VERSION_libgcc-initial ?= "${GCCVERSION}"
PREFERRED_VERSION_nativesdk-gcc ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_nativesdk-libgcc ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_nativesdk-libgcc-initial ?= "${SDKGCCVERSION}"
PREFERRED_VERSION_binutils ?= "${BINUVERSION}"
PREFERRED_VERSION_binutils-native ?= "${BINUVERSION}"
PREFERRED_VERSION_binutils-cross-${TARGET_ARCH} ?= "${BINUVERSION}"
PREFERRED_VERSION_binutils-crosssdk-${SDK_ARCH} ?= "${BINUVERSION}"
PREFERRED_VERSION_binutils-cross-canadian-${TRANSLATED_TARGET_ARCH} ?=
                "${BINUVERSION}"
PREFERRED_VERSION_gdb ?= "${GDBVERSION}"
PREFERRED_VERSION_gdb-cross-${TARGET_ARCH} ?= "${GDBVERSION}"
PREFERRED_VERSION_gdb-cross-canadian-${TRANSLATED_TARGET_ARCH} ?= "${GDBVERSION}"
PREFERRED_VERSION_linux-libc-headers ?= "${LINUXLIBCVERSION}"
PREFERRED_VERSION_nativesdk-linux-libc-headers ?= "${LINUXLIBCVERSION}"
```

```
PREFERRED_VERSION_glibc                    ?= "${GLIBCVERSION}"
PREFERRED_VERSION_glibc-locale             ?= "${GLIBCVERSION}"
PREFERRED_VERSION_glibc-mtrace             ?= "${GLIBCVERSION}"
PREFERRED_VERSION_glibc-scripts            ?= "${GLIBCVERSION}"
PREFERRED_VERSION_nativesdk-glibc          ?= "${GLIBCVERSION}"
PREFERRED_VERSION_glibc-initial            ?= "${GLIBCVERSION}"
PREFERRED_VERSION_nativesdk-glibc-initial  ?= "${GLIBCVERSION}"
PREFERRED_VERSION_cross-localedef-native   ?= "${GLIBCVERSION}"
PREFERRED_VERSION_uclibc                   ?= "${UCLIBCVERSION}"
PREFERRED_VERSION_uclibc-initial           ?= "${UCLIBCVERSION}"
# don't use version earlier than 1.4 for gzip-native, as it's necessary for
# some packages using an archive format incompatible with earlier gzip
PREFERRED_VERSION_gzip-native ?= "1.6"
# Added preferences
BOOSTVERSION ?= "1.57%"
VALGRINDVERSION ?= "3.10%"
PREFERRED_VERSION_boost = "${BOOSTVERSION}"
PREFERRED_VERSION_valgrind = "${VALGRINDVERSION}"
# Setup suitable toolchain flags
require conf/distro/include/as-needed.inc
```

A.1.4 File udoo-image-qt5.bb

```
# This requires the meta-qt5 layer in your bblayers.conf !!!!
# source: http://wiki.wandboard.org/index.php/Building_Qt5_using_yocto_on_Wandboard
# If you plan to use Qt5 eglfs plugin for accelerated graphics using the framebuffer,
# you need to discard X11 and wayland so the proper graphics drivers get included.
# To achieve this add the following to your conf/local.conf :
# DISTRO_FEATURES_remove = "x11 wayland"
DESCRIPTION = "A Qt 5.5.1+ image. Tailored for the UDOO boards"
DEPENDS += "virtual/bootloader"
IMAGE_FEATURES += "splash ssh-server-openssh package-management debug-tweaks"
UDOO_TOOLS = " \
```

```
    ${@bb.utils.contains("MACHINE_FEATURES", "usbhost",
        "packagegroup-base-usbhost", "", d)} i2c-tools resize-rootfs \
    ${@base_conditional("ENABLE_CAN_BUS", "1", "canutils", "", d)} minicom \
"
QT_TOOLS = " \
    qtbase-fonts qtbase-plugins qtbase-tools \
    qtdeclarative qtdeclarative-plugins qtdeclarative-tools \
    qtdeclarative-qmlplugins qtmultimedia \
    qtmultimedia-plugins qtmultimedia-qmlplugins \
    qtsvg qtsvg-plugins qtsensors \
    qtimageformats-plugins qtsystems \
    qtsystems-tools qtsystems-qmlplugins \
    qtscript qt3d qt3d-qmlplugins \
    qtwebkit qtwebkit-examples-examples \
    qtwebkit-qmlplugins qtgraphicaleffects-qmlplugins \
    qtconnectivity-qmlplugins qtlocation-plugins \
    qtlocation-qmlplugins cinematicexperience \
    qt5-env qtserialbus \
"
GSTREAMER_TOOLS = " \
    packagegroup-fsl-gstreamer1.0-full \
"
REMOTE_DEBUGGING = " \
    gdbserver openssh-sftp-server \
"
FSL_TOOLS = " \
    packagegroup-fsl-tools-testapps \
    packagegroup-fsl-tools-benchmark \
    packagegroup-fsl-tools-gpu \
"
IMAGE_INSTALL = "\
    packagegroup-core-boot packagegroup-core-full-cmdline \
```

```
    packagegroup-base ${CORE_IMAGE_EXTRA_INSTALL} \
    ${QT_TOOLS} ${UDOO_TOOLS} ${GSTREAMER_TOOLS} \
    ${REMOTE_DEBUGGING} ${FSL_TOOLS} \
    cairo pango fontconfig freetype pulseaudio dbus \
    alsa-lib alsa-tools alsa-state fsl-alsa-plugins \
"
IMAGE_INSTALL_append = "\
        canutils tree dfu-util libconfig boost \
        eglinfo-x11 glmark2 libsocketcan ppp-dialin ppp \
        iptables usbutils mosquitto c-ares libxml2 \
        sqlite3 valgrind openssh \
"
inherit core-image
# for populate_sdk to create a valid toolchain
inherit populate_sdk_qt5
```

A.1.5 File 0076-Fix-build-with-gcc-6.patch

```
From efdf2b53b907c96ad3f00275588eb311335d0c91 Mon Sep 17 00:00:00 2001
From: Ioan-Adrian Ratiu <adrian.ratiu@ni.com>
Date: Thu, 12 May 2016 15:24:25 +0300
Subject: [PATCH] Fix build with gcc 6
        * Make-lang.in: Invoke gperf with -L C++.
        * cfns.gperf: Remove prototypes for hash and libc_name_p
        inlines.
        * cfns.h: Regenerated.
        * except.c (nothrow_libfn_p): Adjust.
svn rev: r233572
Upstream-status: Backport [gcc 4.9]
Signed-off-by: Ioan-Adrian Ratiu <adrian.ratiu@ni.com>
--
 gcc/cp/Make-lang.in |  2 +-
 gcc/cp/cfns.gperf   | 10 ++----
 gcc/cp/cfns.h       | 41 +++++++++++++++--------------
```

```
 gcc/cp/except.c     |  3 ++-
 4 files changed, 19 insertions(+), 37 deletions(-)
diff -git a/gcc/cp/Make-lang.in b/gcc/cp/Make-lang.in
index bd1c1d7..a0ea0d4 100644
-- a/gcc/cp/Make-lang.in
+++ b/gcc/cp/Make-lang.in
@@ -111,7 +111,7 @@ else
 # deleting the $(srcdir)/cp/cfns.h file.
 $(srcdir)/cp/cfns.h:
 endif
-       gperf -o -C -E -k '1-6,$$' -j1 -D -N 'libc_name_p' -L ANSI-C \
+       gperf -o -C -E -k '1-6,$$' -j1 -D -N 'libc_name_p' -L C++ \
                $(srcdir)/cp/cfns.gperf -output-file $(srcdir)/cp/cfns.h
#
diff -git a/gcc/cp/cfns.gperf b/gcc/cp/cfns.gperf
index 05ca753..d9b16b8 100644
-- a/gcc/cp/cfns.gperf
+++ b/gcc/cp/cfns.gperf
@@ -1,3 +1,5 @@
+%language=C++
+%define class-name libc_name
 %{
 /* Copyright (C) 2000-2014 Free Software Foundation, Inc.
@@ -16,14 +18,6 @@ for more details.
 You should have received a copy of the GNU General Public License
 along with GCC; see the file COPYING3.  If not see
 <http://www.gnu.org/licenses/>.  */
-#ifdef __GNUC__
-__inline
-#endif
-static unsigned int hash (const char *, unsigned int);
-#ifdef __GNUC__
```

```
-__inline
-#endif
-const char * libc_name_p (const char *, unsigned int);
 %}
 %%
 # The standard C library functions, for feeding to gperf; the result is used
diff -git a/gcc/cp/cfns.h b/gcc/cp/cfns.h
index c845ddf..65801d1 100644
-- a/gcc/cp/cfns.h
+++ b/gcc/cp/cfns.h
@@ -1,5 +1,5 @@
-/* ANSI-C code produced by gperf version 3.0.3 */
-/* Command-line: gperf -o -C -E -k '1-6,$' -j1 -D -N libc_name_p
                         -L ANSI-C cfns.gperf  */
+/* C++ code produced by gperf version 3.0.4 */
+/* Command-line: gperf -o -C -E -k '1-6,$' \
        -j1 -D -N libc_name_p -L C++ -output-file cfns.h cfns.gperf  */
 #if !((' ' == 32) && ('!' == 33) && ('"' == 34) && ('#' == 35) \
    && ('%' == 37) && ('&' == 38) && ('\" == 39) && ('(' == 40) \
@@ -28,7 +28,7 @@
 #error "gperf generated tables don't work with this execution character set.
 Please report a bug to <bug-gnu-gperf@gnu.org>."
 #endif
-#line 1 "cfns.gperf"
+#line 3 "cfns.gperf"
 /* Copyright (C) 2000-2014 Free Software Foundation, Inc.
@@ -47,25 +47,18 @@ for more details.
 You should have received a copy of the GNU General Public License
 along with GCC; see the file COPYING3.  If not see
 <http://www.gnu.org/licenses/>.  */
-#ifdef __GNUC__
-__inline
```

```
-#endif
-static unsigned int hash (const char *, unsigned int);
-#ifdef __GNUC__
-__inline
-#endif
-const char * libc_name_p (const char *, unsigned int);
 /* maximum key range = 391, duplicates = 0 */
-#ifdef __GNUC__
-__inline
-#else
-#ifdef __cplusplus
-inline
-#endif
-#endif
-static unsigned int
-hash (register const char *str, register unsigned int len)
+class libc_name
+{
+private:
+  static inline unsigned int hash (const char *str, unsigned int len);
+public:
+  static const char *libc_name_p (const char *str, unsigned int len);
+};
+
+inline unsigned int
+libc_name::hash (register const char *str, register unsigned int len)
 {
   static const unsigned short asso_values[] =
     {
@@ -122,14 +115,8 @@ hash (register const char *str, register unsigned int len)
   return hval + asso_values[(unsigned char)str[len - 1]];
 }
```

```
-#ifdef __GNUC__
-__inline
-#ifdef __GNUC_STDC_INLINE__
-__attribute__ ((__gnu_inline__))
-#endif
-#endif
 const char *
-libc_name_p (register const char *str, register unsigned int len)
+libc_name::libc_name_p (register const char *str, register unsigned int len)
 {
   enum
     {
diff -git a/gcc/cp/except.c b/gcc/cp/except.c
index 221971a..32340f5 100644
-- a/gcc/cp/except.c
+++ b/gcc/cp/except.c
@@ -1030,7 +1030,8 @@ nothrow_libfn_p (const_tree fn)
      unless the system headers are playing rename tricks, and if
      they are, we don't want to be confused by them.  */
   id = DECL_NAME (fn);
-  return !!libc_name_p (IDENTIFIER_POINTER (id), IDENTIFIER_LENGTH (id));
+  return !!libc_name::libc_name_p (IDENTIFIER_POINTER (id),
+                                   IDENTIFIER_LENGTH (id));
 }
 /* Returns nonzero if an exception of type FROM will be caught by a
-
2.8.2
```

A.1.6 File libUtility_1.0.0.bb

```
SUMMARY = "libUtility"
SECTION = "libs"
LICENSE = "GPL-2.0"
LIC_FILES_CHKSUM = "file://Readme;md5=793b3fc16d7acd3016eb41a862142a08"
```

```
DEPENDS = "boost libconfig"
PN = "libUtility"
S = "${WORKDIR}/${PN}_${PV}"
FILESPATH = "${FILE_DIRNAME}/${PN}_${PV}"
SRC_URI = "file://${PN}_${PV}.tar.bz2"
EXTRA_OECMAKE += "-DCMAKE_VERBOSE_MAKEFILE=0"
TARGET_CXXLAGS += "-DMMLOGGER"
inherit cmake
```

A.2 Configurazione Toaster

A.2.1 File toaster.conf

```
Alias /static /var/www/toaster/static_files
<Directory /var/www/toaster/static_files>
  Order allow,deny
  Allow from all
  Require all granted
</Directory>
WSGIDaemonProcess toaster_wsgi \
python-path=/var/www/toaster/sources/poky/bitbake/lib/toaster:\
/var/www/toaster/venv/lib/python2.7/site-packages
WSGIScriptAlias / "/var/www/toaster/sources/poky/bitbake/lib/toaster/toastermain/wsgi.py"
<Location />
  WSGIProcessGroup toaster_wsgi
</Location>
```

A.2.2 File runbuilds-service.sh

```
#!/bin/bash
$ cd /var/www/toaster
$ source ./venv/bin/activate
$ (venv) MACHINE=udooqdl source ./sources/setup-environment build
```

```
$ (venv) source ../sources/poky/bitbake/bin/toaster $1 noweb
```

A.2.3 File runbuilds.service

```
[Unit]
Description=Toaster runbuilds
[Service]
Type=forking
User=mm-develop
ExecStart=/usr/bin/screen -d -m -S runbuilds \
/var/www/toaster/sources/poky/bitbake/lib/toaster/runbuilds-service.sh start
ExecStop=/usr/bin/screen -S runbuilds -X quit
WorkingDirectory=/var/www/toaster/sources/poky
[Install]
WantedBy=multi-user.target
```

Bibliografia

[1] https://ec.europa.eu/transport/themes/its_en
[2] http://www.etsi.org/
[3] https://standards.ieee.org/develop/wg/1609.html
[4] http://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.02.00_20/en_302663v010200a.pdf
[5] https://www.arib.or.jp/english/
[6] https://www.arib.or.jp/english/html/overview/doc/5-STD-T109v1_0-E1.pdf
[7] http://www.magnetimarelli.com/it/azienda
[8] https://www.udoo.org/udoo-dual-and-quad/
[9] http://www.3gpp.org/
[10] http://free-electrons.com/doc/training/embedded-linux/embedded-linux-slides.pdf
[11] https://www.gnu.org/software/make/
[12] https://www.gnu.org/software/automake/manual/html_node/index.html
[13] https://cmake.org/
[14] http://crosstool-ng.github.io/
[15] http://www.pengutronix.de/en/software/ptxdist
[16] https://buildroot.org/downloads/manual/manual.pdf
[17] http://www.yoctoproject.org/
[18] https://www.openembedded.org/wiki/Main_Page
[19] https://source.android.com/devices/architecture/dto/
[20] http://www.yoctoproject.org/docs/2.1.3/ref-manual/ref-manual.html
[21] http://www.yoctoproject.org/docs/2.1.3/sdk-manual/sdk-manual.html
[22] http://www.yoctoproject.org/docs/2.1.3/toaster-manual/toaster-manual.html

[23] https://www.djangoproject.com/
[24] https://httpd.apache.org/
[25] https://www.mysql.com/it/
[26] http://www.yoctoproject.org/docs/2.1.3/dev-manual/dev-manual.html
[27] http://www.3gpp.org/release-14
[28] http://ieeexplore.ieee.org/document/4526014/?part=1