Usporedba razvojnih okolina za Internet stvari

Filinić, Goran

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Goran Filinić

USPOREDBA RAZVOJNIH OKOLINA ZA INTERNET STVARI

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Goran Filinić

Matični broj: 0016124457

Studij: Baze podataka i baze znanja

USPOREDBA RAZVOJNIH OKOLINA ZA INTERNET STVARI

DIPLOMSKI RAD

Mentor:

prof. dr. sc. Ivan Magdalenić

Varaždin, lipanj 2024.

Goran Filinić

Izjava o izvornosti

Izjavljujem da je ovaj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

Diplomski rad istražuje i uspoređuje razvojne okoline za razvoj projekata u području Interneta stvari (IoT-a). Fokus se stavlja na analizu karakteristika i funkcionalnosti različitih razvojnih okolina, s posebnim naglaskom na Yocto Project i Buildroot. Cilj istraživanja je razumijevanje procesa i postupaka razvoja projekata u ovim okolinama te ocjenjivanje njihovih prednosti i nedostataka. U praktičnom dijelu rada projekt implementira odabrane funkcionalnosti IoT uređaja, uz upravljanje i izvještavanje uređajima putem grafičkog sučelja. Kao primjer praktične implementacije gradi se kernel image za Raspberry Pi uređaj. Ovaj rad pruža korisne uvide u proces razvoja projekata u području IoT te pruža jedno od mogućih rješenja optimalne razvojne okoline za konkretne potrebe na tržištu.

Ključne riječi: internet stvari, IoT, kernel image, Raspberry Pi, razvojne okoline, ugradbeni sustavi, yocto project

Sadržaj

1. Uvod

U diplomskom radu prikazuju se aktualne metodologije, tehnologije i rješenja koja su povezana s ugradbenim sustavima te uređajima interneta stvari (IoT). Kroz rad se uspoređuju se razne razvojne okoline, njihove prednosti i mane. Detaljno se obrađuju razvojne okoline Yocto Project i Buildroot te na njihove različite primjene i sposobnosti. U praktičnom dijelu rada koristi se razvojna okolina Yocto Project kako bi se razvio prilagođeni operacijski sustav s grafičkom aplikacijom koja prikazuje podatke prikupljene s povezanim DHT22 senzorom temperature i vlažnosti.

Cilj diplomskog rada je prikazati detaljnu usporedbu dviju razvojnih okolina, Buildroot-a i Yocto Project-a, s naglaskom na njihove specifične značajke, prednosti i mane u kontekstu razvoja ugradbenih sustava i IoT uređaja. Predmet istraživanja bazira se na procesima IoT-a te relevantne infrastrukture razvojnih okolina.

Rad se sastoji od sedam poglavlja:

1. U uvodnom dijelu rada prikazuju se predmet, cilj istraživanja te struktura rada. Također se ukratko opisuje tema i sadržaj diplomskog rada.

2. Metode i tehnike rada prikazuju znanstvene metode korištene u radu, kako u teorijskom, tako i u praktičnom dijelu rada.

3. Proces razvoja interneta stvari obrađuje razvojni proces IoT-a, uključujući koncepte ugradbenih sustava.

4. Razvojna infrastruktura za ugradbene sustave se fokusira na razvojnu infrastrukturu potrebnu za ugradbene sustave. Opisuje se Linux operacijski sustav i alati za križno kompiliranje, koji su ključni za razvoj ovih sustava.

5. Detaljna usporedba dviju odabranih razvojnih okolina detaljno analiziraja i uspoređuje dvije razvojne okoline – Buildroot i Yocto Project. Pruža se uvid u njihove prednosti, nedostatke i specifične karakteristike.

6. Praktična implementacija Yocto projekta za Raspberry Pi s grafičkim sučeljem prikazuje praktičnu implementaciju Yocto projekta na Raspberry Pi s grafičkim sučeljem. Obuhvaća opis projektne strukture, konfiguracijskih datoteka, aplikacijskog sloja, recepata za slike operacijskog sustava te biblioteka razvijenog operacijskog sustava i aplikacija.

7. U završnom poglavlju objašnjava se važnost istraživane tematike za poslovanje i budućnost te se zaokružuje cjelokupni rad.

2. Metode i tehnike rada

U svrhu istraživanja razvoja IoT sustava odabirani su različiti alati i metode. Za teorijski dio rada korištene su metode:

- **Sekundarno istraživanje**: Odnosi se na prikupljanje i analizu podataka koji su prethodno bili dostupni na raznim izvorima (stručni članci, knjige, repozitoriji i slično).
- **Komparativna analiza**: Usporedba razvojnih okolina Buildroot i Yocto radi identificiranja njihovih prednosti i mana te odabira najprikladnije okoline za specifične potrebe projekta.
- **Eksperimentalna metoda**: Praktična implementacija i testiranje razvijenih rješenja na stvarnim hardverskim platformama poput Raspberry Pi-a.

Za praktični dio rada korišteno je sljedeće:

- **Git**: Verzijski kontrolni sustav korišten za praćenje promjena u kodu, omogućujući kolaboraciju i upravljanje verzijama projekta.
- **Yocto (Bitbake)**: Yocto Project, zajedno s Bitbake alatom, korišten je za generiranje prilagođenih Linux distribucija.
- **Qt Creator**: Integrirano razvojno okruženje korišteno za razvoj grafičke aplikacije koja prikazuje podatke senzora. Qt Creator podržava razvoj s Qt frameworkom.
- **Linux Terminal**: Terminal je korišten za izvršavanje različitih komandi potrebnih za upravljanje sustavom.

U ovom poglavlju treba opisati koje će metode i tehnike biti korištene pri razradi teme, kako su provedene istraživačke aktivnosti, koji su programski alati ili aplikacije korišteni.

3. Proces razvoja interneta stvari

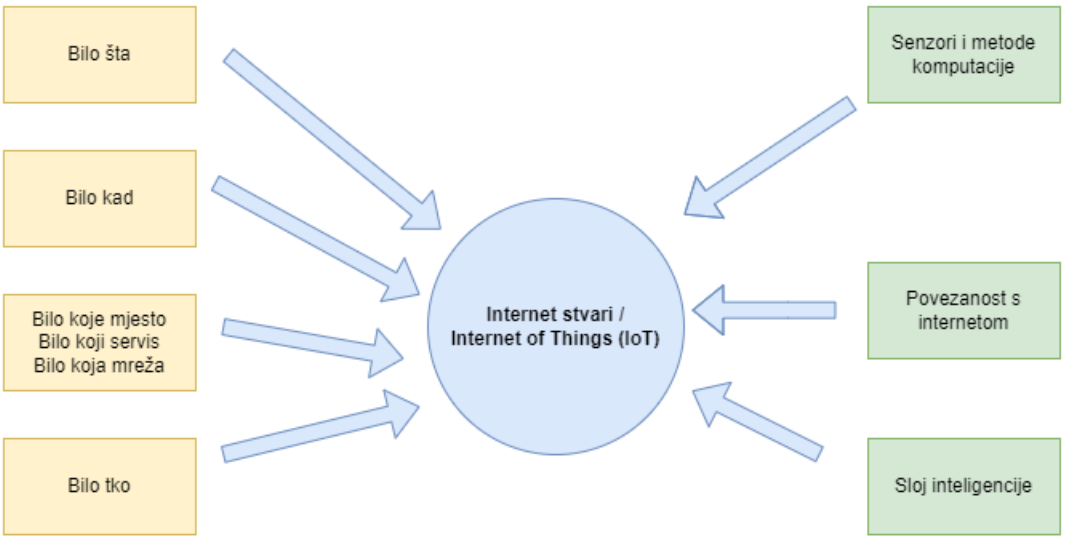
Kako bi se mogao razumjeti proces razvoja interneta stvari, potrebno je opisati i definirati zahtjeve ugradbenih sustava na kojima se temelji internet stvari.

3.1. Ugradbeni sustavi

Ugradbeni sustav (*eng. embedded system*) odnosi se na elektroničku opremu s računalnom jezgrom (najčešće mikrokontroleri) koja je, za razliku od osobnog računala, dizajnirana s ciljem da zadovolji točno određenu funkciju. Obično je optimizirana da zadovolji stroge zahtjeve obrade, pouzdanosti, potrošnje energije, veličine i troškova [1]. Za razliku od računala opće namjene koja imaju mnogo više resursa za obradu na raspolaganju, ugradbeni sustavi su ograničeni u veličini, snazi obrade te kapacitetu memorije. U ovu skupinu tako spadaju sustavi za praćenje i upravljanje avionima, većina modernih kućnih aparata, ali i medicinski instrumenti. Primjetno je da je velika širina primjene, zbog toga se individualni sustavi mogu znatno razlikovati i po mehaničkim komponentama i po softveru.

Iz ovog razloga ne postoji jedan unificiran način klasifikacije ovih sustava, ali jedna opća podjela bila bi:

- **Ugradbeni sustavi u pravom vremenu** - ovo su sustavi koji čine sržnu komponentu sustava koji zahtijevaju brze i precizne odgovore na vanjske događaje i podražaje. Oni imaju kritičnu vremensku komponentu te moraju biti otporni na greške i pravovremeno reagirati kako bi održali sigurnost i funkcionalnost. Primjeri takvih sustava uključuju upravljanje letom aviona, upravljanje vozilima autonomne vožnje te medicinske instrumente koji zahtijevaju hitno i precizno djelovanje ili obavještavanje u kritičnim situacijama. Ovi sustavi najčešće imaju pravovremeni operacijski sustav koji omogućava brze i pouzdane izračune te upravljanje resursima u skladu s vremenskim ograničenjima, ovi operacijski sustavi bit će objašnjeni u poglavlju 4.3.1 .
- **Opći ugradbeni sustavi** - ovi sustavi obuhvaćaju širok spektar primjena u različitim industrijama i svakodnevnom životu. Za razliku od ugradbenih sustava u pravom vremenu, ovi sustavi obično ne zahtijevaju tako stroge zahtjeve vremenske determinističnosti. Primjeri uključuju kućanske aparate, audio i video sustave, industrijske strojeve, i slično. Opći ugradbeni sustavi mogu koristiti različite vrste operacijskih sustava, ovisno o njihovim zahtjevima i namjeni. Neki od njih koriste realno-vremenske operacijske sustave za određene zadatke gdje je potrebno osigurati brzi odziv, dok drugi mogu koristiti generalne operacijske sustave poput *Linuxa*.



Slika 1: Koncept interneta stvari; Vlastiti rad baziran na [2]

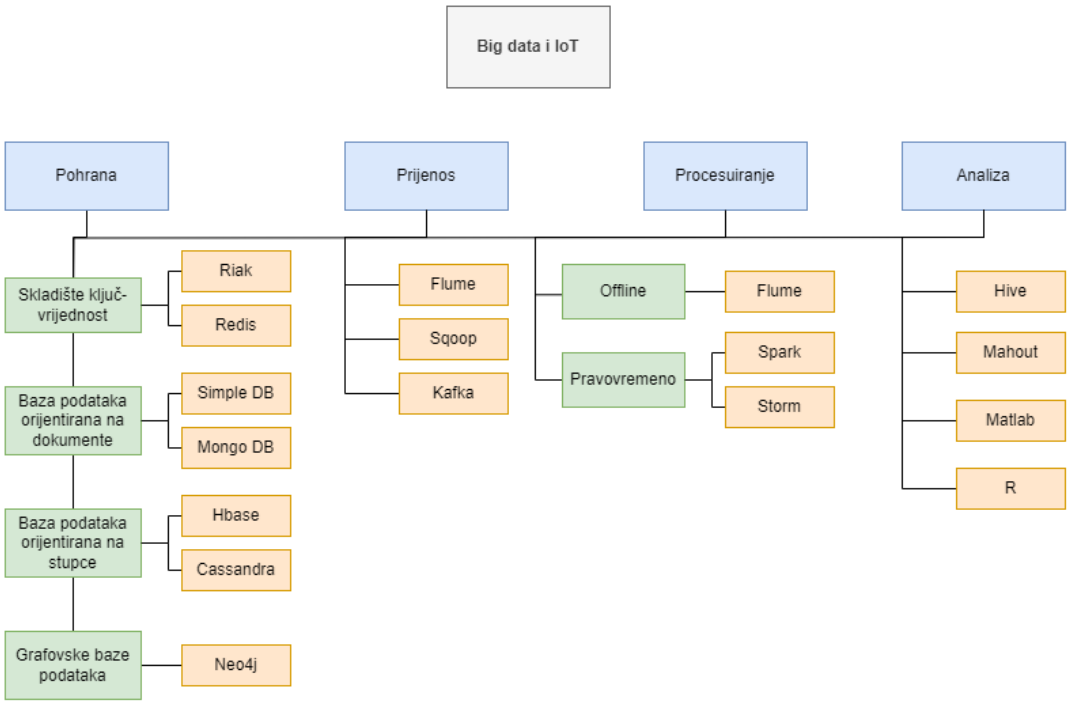
3.2. Internet stvari

Internet stvari je mreža međusobno povezanih objekata. U praksi internet stvari zapravo je mreža manjih mreža koja međusobno čine jednu cjelinu. Tako na primjer moderni automobil ima mrežu ugradbenih sustava koja služe za praćenje različitih komponenta automobila, povezana s kontrolerom za upravljanje temperaturom te informativno-zabavnim sustavom (*eng. infotainment system*) s kojim je korisnik u interakciji. Nije čudna pojava i da korisnik može preko interneta pristupiti nekim opcijama. S obzirom na to da automobil mora ima pristup internetu kako bi npr. izračunao rutu za navigaciju tako je sustav auta zapravo dio većeg sustava interneta stvari. Slika 1 prikazuje osnovni koncept interneta stvari, prikazujući raznolike elemente koji ga čine. Na lijevoj strani slike su prikazani aspekti koji naglašavaju univerzalnu povezanost i fleksibilnost interneta stvari, dok desna strana slike ističe ulogu senzora, povezanosti s internetom te sloj inteligencije u kontekstu interneta stvari.

Internet stvari može se definirati kao dinamična globalna mrežna infrastruktura koja omogućuje identifikaciju, kontrolu i praćenje svakog objekta na Zemlji putem interneta prema određenom protokolu sporazuma, te putem međusobne povezanosti fizičkih i virtualnih stvari temeljenih na interoperabilnosti informacijskih i komunikacijskih tehnologija [2].

Danas se koncept interneta stvari (*eng. Internet of Things - IoT*) sve više povezuje s velikim podacima, jer rastući broj IoT uređaja generira ogromne količine podataka koji zahtijevaju posebne arhitekture i tehnologije za njihovo upravljanje i analizu. Ovi podaci, poznati kao "big data", obuhvaćaju širok spektar informacija i zahtijevaju sofisticirane metode obrade i analize radi dobivanja korisnih uvida i potpore donošenju odluka. Slika 1 slikovito prikazuje komponente konceptualizacije interneta stvari.

Ovi podaci se prikupljaju putem različitih komunikacijskih protokola, odnosno rješenja, poput WiFi-ja, ZigBee-ja, Bluetootha i GSM-a, te se prenose kroz mrežu IoT infrastrukture. Taj proces omogućuje kontinuirano prikupljanje podataka iz različitih izvora, stvarajući ogromne



Slika 2: Popularni alati za implementaciju IoT-a i big data tehnologije; Vlastiti rad baziran na [2]

količine podataka koje karakterizira visoka razina volumena, brzine, raznolikosti i vjerodostojnosti. Uz sve veći broj pametnih objekata i rastuću potražnju za njihovim uslugama, integracija IoT-a s velikim podacima postaje važna za različita područja primjene, uključujući pametne gradove, zdravstvo, upravljanje katastrofama i zaštitu okoliša. Ovo sveukupno stvara potrebu za razvojem i primjenom inovativnih tehnologija i pristupa koji će omogućiti učinkovito upravljanje ogromnim količinama podataka te izvlačenje vrijednih uvida za poboljšanje kvalitete života i podršku odlučivanju [2].

3.2.1. Popularni alati za implementaciju IoT-a i big data tehnologije

Slika 2. prikazuje popularne alate za implementaciju raznih aspekata IoT tehnologije i big data analize. U području pohrane podataka, koriste se različite vrste baza podataka kako bi se efikasno upravljalo velikim količinama podataka. To uključuje skladište ključ-vrijednost kao što su Riak i Redis, baze podataka orijentirane na dokumente poput Simple DB i Mongo DB, baze podataka orijentirane na stupce kao što su Hbase i Cassandra, te grafovske baze podataka kao što je Neo4j. Za prijenos podataka, popularni alati uključuju Flume, Sqoop i Kafka, koji omogućuju učinkovit prijenos podataka između različitih izvora i odredišta.

Procesuiranje podataka u IoT-u obuhvaća offline i realno-vremenske metode. Offline metode poput MapReduce pružaju mogućnost za obradu velikih količina podataka u određenim vremenskim intervalima. S druge strane, za pravovremeno procesuiranje podataka koriste se alati poput Sparka i Storma, koji omogućuju obradu podataka u realnom vremenu, čime se osigurava brza analiza podataka na brzim tokovima podataka. Za analizu podataka, popularni alati uključuju Hive, Mahout, Matlab i R. Ovi alati omogućuju dubinsku analizu podataka, izvla-

čenje uzoraka i donošenje informiranih odluka na temelju analize podataka iz IoT okruženja.

Sve spomenute tehnologije omogućene su proširenjem komunikacije koje donosi Web 2.0, a dalje je potaknuto integracijom različitih alata i aplikacija. Sada je moguće integrirati heterogene izvore prikupljanja podataka s moćnim alatima za analizu velikih podataka te vizualizacijskim aplikacijama . Takve mogućnosti imaju obećavajuće posljedice za razvoj aplikacija za pametne infrastrukture odnosno mreže interneta stvari.

3.2.2. Benefiti interneta stvari i velike količine podataka

Integracija interneta stvari (IoT) s tehnologijama velikih podataka (big data) donosi brojne koristi i mogućnosti za različite sektore i područja primjene

3.2.2.1. Heterogeni izvori i skladištenje podataka

Integracijom heterogenih izvora podataka, poput senzora, daljinskih senzora, kamera i pametnih telefona, s tehnologijama velikih podataka koje pomažu u upravljanju podacima (kao što su prikupljanje podataka, pohrana podataka, obrada podataka i analiza podataka), omogućuje se značajno povezivanje različitih izvora podataka kako bi se isporučile korisne nove informacije i uvidi. Podaci se mogu prikupljati s IoT senzora kao i s drugih izvora kako bi se omogućila izgradnja prediktivnih modela. Stoga, s rastućom dostupnošću bogatih novih izvora podataka, korištenje velikih podataka i interneta stvari može donijeti zanimljive rezultate u aplikacijama za pametno okruženje.

3.2.2.2. Povezivanje

Povezivanje između međusobno povezanih podatkovnih čvorova i sustava za upravljanje podacima služi kao osnova koja osigurava operativni uspjeh. Stoga, budući da je danas dostupno mnogo vrsta komunikacijskih tehnologija, predložene arhitekture u aplikacijama za pametno okruženje moraju biti fleksibilne kako bi se nosile s različitim komunikacijskim protokolima, kako udaljenim tako i lokalnim. Kada se koriste IoT uređaji i slični izvori podataka, imati infrastrukturu s učinkovitim mogućnostima pohrane može povećati učinkovitost i učinkovitost obrade podataka te poboljšati dizajn i funkcionalnost aplikacija. Nadalje, na temelju tehnologija analize velikih podataka, postaje moguće omogućiti obradu podataka s niskom latencijom uz održavanje potrebnih struktura pohrane za velike skupove podataka na jeftinom komoditetom hardveru.

3.2.2.3. Analiza u realnom vremenu

U mnogim aplikacijama za okoliš, poput upravljanja katastrofama, sustavi rane upozorenja temeljeni na analizi u stvarnom vremenu ključni su zahtjev, ali i oni dolaze s izazovima. Povezanost između različitih izvora podataka često rezultira visokom brzinom generiranja ogromnih količina podataka, stvarajući dinamičnu i zahtjevnu situaciju koja izaziva performanse

sustava za analizu podataka u stvarnom vremenu. Za suočavanje s ovim izazovom i omogućavanje obrade u stvarnom vremenu, potrebna su rješenja softvera i tehnološke sposobnosti posvećene tim procesima. Veliki podaci danas omogućuju takva rješenja, koja mogu biti važna u donošenju odluka za vrijeme i učinkovite aplikacije za hitne slučajeve.

3.2.2.4. Ekonomičnost

Mnoge tehnologije velikih podataka otvorenog su kôda, što omogućuje važno smanjenje troškova potrebnih za razvoj i implementaciju novih aplikacija. Ekonomičnost je posebno važna u zemljama u razvoju, gdje nedostatak dovoljnih sredstava može spriječiti implementaciju sustava za upravljanje

3.2.3. Moderni izazovi interneta stvari

Kako bi se bolje objasnile razvojne okoline za razvoj IoT, potrebno je analizirati moderne izazove s kojima se industrija suočava. Izazove možemo podijeliti prema paradigmi utjecaja [3]:

- **Poslovna**
- **Društvena**
- **Tehnološka**

3.2.3.1. Poslovni izazovi

U kontekstu poslovnih izazova, važno je razumjeti kako IoT može pružiti vrijednost različitim sektorima i klijentima. Razvoj održivog poslovnog modela za IoT ključan je kako bi se izbjegla ponovna pojava "balona" tehnološke industrije. Taj model mora zadovoljiti sve zahtjeve za sve vrste e-trgovine: vertikalne tržišta, horizontalna tržišta i potrošačka tržišta. Međutim, ova kategorija uvijek je izložena regulatornom i pravnom nadzoru.

Pružatelji rješenja "od kraja do kraja" koji posluju u vertikalnim industrijama i pružaju usluge koristeći analitiku u oblaku bit će najuspješniji u monetizaciji velikog dijela vrijednosti u IoT-u. Iako mnoge IoT aplikacije mogu privući skromne prihode, neke mogu privući više. Bez značajnog opterećenja postojeće komunikacijske infrastrukture, operatori imaju potencijal otvaranja značajnog izvora novih prihoda koristeći IoT tehnologije.

Važno je razumjeti vrijednosni lanac i poslovni model za IoT aplikacije u svakoj kategoriji IoT-a. IoT se može podijeliti u tri kategorije na temelju upotrebe i baze klijenata [3]:

- **Industrijski IoT**: Ovaj podskup obuhvaća stvari poput povezanih električnih brojila, sustava za obradu otpadnih voda, mjerača protoka, monitora cjevovoda, robotske proizvodnje i drugih vrsta povezanih industrijskih uređaja i sustava.
- **Komercijalni IoT**: Ova kategorija obuhvaća stvari poput kontrole inventara, praćenja uređaja i povezanih medicinskih uređaja.

- **Potrošački IoT:** Ova kategorija uključuje povezane uređaje poput pametnih automobila, telefona, satova, prijenosnih računala, povezanih kućanskih aparata i drugih zabavnih sustava.

3.2.3.2. Društveni izazovi

IoT stvara jedinstvene izazove za privatnost, mnoge od njih izlaze izvan problema privatnosti podataka koji trenutno postoje. Veliki dio toga proizlazi iz integriranja uređaja u okolinu bez svjesnog individualnog korištenja.

Ovo postaje sve učestalije kod potrošačkih uređaja, poput praćenja uređaja za telefone i automobile, kao i pametnih televizora. U pogledu potonjeg, funkcije prepoznavanja glasa ili slike se integriraju i mogu kontinuirano slušati razgovore ili nadgledati aktivnost te selektivno prenositi te podatke u oblak za obradu, što ponekad uključuje i treću stranu. Prikupljanje ovih informacija izlaže pravne i regulatorne izazove koji se odnose na zakon o zaštiti podataka i privatnosti. Osim toga, mnogi scenariji IoT-a uključuju postavljanje uređaja i aktivnosti prikupljanja podataka s multinacionalnim ili globalnim djelovanjem koji prelaze društvene i kulturne granice. Kako bi se ostvarile prilike IoT-a, potrebno je razviti strategije za poštovanje individualnih izbora privatnosti na širokom spektru očekivanja, dok se istovremeno potiče inovacija u novim tehnologijama i uslugama.

Regulativni standardi za tržišta podataka nedostaju posebno za posrednike podataka; to su tvrtke koje prodaju podatke prikupljene s različitih izvora. Iako podaci izgledaju kao valuta IoT-a, postoji nedostatak transparentnosti o tome tko dobiva pristup podacima i kako se ti podaci koriste za razvoj proizvoda ili usluga i prodaju oglašivačima i trećim stranama. Potrebne su jasne smjernice o zadržavanju, korištenju i sigurnosti podataka, uključujući metapodatke (podaci koji opisuju druge podatke).

Ostali društveni problemi uključuju [3]:

1. Promjene u zahtjevima potrošača: Zahtjevi i potrebe potrošača neprestano se mijenjaju.
2. Brzi razvoj novih uređaja i primjena: Novi uređaji i nove primjene IoT tehnologije nastaju i rastu brzinom svjetlosti.
3. Troškovi i resursi za integraciju novih značajki: Inventiranje i reintegracija značajki i sposobnosti koje su neophodne predstavljaju troškove i zahtijevaju vrijeme i resurse.
4. Ekspanzija i promjena upotrebe IoT tehnologije: Upotreba IoT tehnologije neprestano se širi i mijenja, često u nepoznatim vodama.
5. Povjerenje potrošača: Svaki od ovih problema može narušiti želju potrošača za kupnjom povezanih proizvoda, što bi spriječilo IoT da ostvari svoj pravi potencijal.
6. Nedostatak razumijevanja ili obrazovanja potrošača o najboljim praksama sigurnosti IoT uređaja: Na primjer, promjena zadanog lozinke IoT uređaja.

3.2.4. Tehnološki izazovi

Postoje mnogi tehnološki izazovi u kontekstu interneta stvari, počevši od konekcije. Naime, tehnološki je izazov povezan s povezivanjem i održavanjem stabilnih veza između velikog broja uređaja. To bi potencijalno mogao biti jedan od važnijih izazova u budućnosti interneta stvari, a predviđa se da će izazvati narušavanje postojeće strukture trenutnih modela komunikacije [2]. Trenutno se koristi centralizirani server/klijent paradigma za autentifikaciju, autorizaciju i povezivanje nekoliko čvorova u mreži, što utječe na sljedeći problematičan izazov sigurnosti.

Internet stvari već je izazvao ozbiljne sigurnosne probleme koji muče različite tvrtke u javnom i privatnom sektoru diljem svijeta. Masivan broj novih čvorišta dodaje se mrežama i internetu, što će omogućiti napadačima nebrojene vektore napada kako bi prodrli u sustav, posebno budući da znatan broj njih pati od sigurnosnih nedostataka. Hakerski napadi na kamere i različite nadzorne uređaje i sustave predstavljaju noćnu moru za sigurnost koju uzrokuje budućnost interneta stvari.

Usko povezano sa sigurnošću su i regulativni standardi. Brzi rast interneta stvari doveo je do različitih tehnologija koje se natječu za standard, što može izazvati buduće probleme osiguranja kompatibilnosti među njima. Problemi kompatibilnosti zahtijevat će razvoj i implementaciju dodatne hardverske i softverske opreme kako bi se osiguralo da se uređaji različitih proizvođača i industrija mogu povezati.

Osim toga, problem predstavlja i dugovječnost i kompatibilnost uređaja. Ubrzani razvoj tehnologije može izazvati buduće probleme osiguranja kompatibilnosti među njima, također i softver može lako postati zastarjeli. Neki od tih tehnologija na kraju će postati zastarjeli u sljedećih nekoliko godina, što efektivno čini beskorisnima uređaje koji ih implementiraju. Ovo je posebno važno jer se uređaji interneta stvari obično koriste mnogo godina, za razliku od generičkih računalnih uređaja koji imaju vijek trajanja od nekoliko godina, pa bi trebali funkcionirati čak i kada proizvođač prestane pružati podršku.

Također, prije spomenuti inteligentni sloj ima niz svojih problema. Tablica 1 prikazuje glavne izazove s kojima se suočavaju big data i IoT u pametnim aplikacijama. Ovi izazovi obuhvaćaju različite aspekte poput volumena podataka, njihove vrijednosti, raznolikosti, vjerodostojnosti, sigurnosti i drugih. Ova tablica pruža pregled ključnih problema koji zahtijevaju pažljivo upravljanje kako bi se osiguralo uspješno implementiranje i korištenje tehnologija pametnih aplikacija.

Tablica 1: Izazovi s kojima se suočavaju big data i IoT u pametnim aplikacijama [2]

Izazov	Opis
Volumen podataka	Opisuje količinu podataka i informacija koje imamo. Ova količina podataka mjerena je u gigabajtima (GB) i eksponencijalno se povećava do zetabajta (ZB) i jotabajta (YB). IoT pridonosi značajnom porastu podataka, što predstavlja srž onoga što big data znači.
Vrijednost podataka	Odnosi se na vrijednost podataka koji se izvlače. Važno je razumjeti troškove i koristi koje se mogu dobiti nakon prikupljanja i analize podataka obilježenih svojim volumenom, brzinom i raznolikošću, što zahtijeva vrijeme i resurse.
Raznolikost podataka	Predstavlja ključni izazov s kojim se suočava IoT i big data općenito, a posebno postaje relevantno u području ekoloških aplikacija. Podaci mogu biti strukturirani (npr. tekst), polustrukturirani (npr. XML i JSON dokument) ili nestrukturirani (npr. slika ili video), što nije jednostavan zadatak za organizaciju takvih različitih tipova podataka.
Vjerodostojnost podataka i modeliranje neizvjesnosti	Specificira razinu kvalitete podataka, odnosno koliko su točni i primjenjivi. To uključuje buku, pristranost i abnormalnosti u podacima. Vjerodostojnost u analizi podataka predstavlja najvažniji izazov u usporedbi s drugim V-ima (tj. volumenom, raznolikošću i brzinom).
Pohrana podataka	Odnosi se na operacije potrebne za izgradnju infrastrukture IoT-a, uključujući mrežne protokole, agregaciju podataka i komunikacijske protokole.
Analiza podataka	Odnosi se na sposobnost analize podataka kako bi se iz njih izvukli korisni uvidi.
Analiza podataka u stvarnom vremenu	Odnosi se na analizu podataka odmah nakon što postanu dostupni, uključujući obradu pomaknutih izvora podataka i upravljanje velikim količinama podataka kako bi se u kraćem vremenu dobile odgovori i uvidi.
Vizualizacija podataka	Odnosi se na implementaciju novih tehnika vizualizacije (npr. grafikoni i dijagrami) radi učinkovitog isticanja odnosa unutar podataka na bolji način.
Sigurnost i privatnost	Odnosi se na izazove privatnosti i sigurnosti koji su značajna prepreka za IoT, s obzirom na rizik od napada i hakiranja povjerljivih informacija koji je vrlo čest zbog raznolikosti korištenih uređaja.
Komunikacija i interoperabilnost	Odnosi se na sposobnost kombiniranja, povezivanja i upravljanja najmanje dvije skupine podataka te na izazove koji se javljaju pri usklađivanju različitih vrsta podataka.
Složenost arhitekture	Odnosi se na povezanost, implementaciju, potrošnju energije i toleranciju na greške u arhitekturi IoT-a, što predstavlja dodatne izazove u izgradnji i održavanju infrastrukture.

3.3. Okvir procesa razvoja interneta stvari

Proces razvoja u području Interneta stvari (IoT) uvelike ovisi o korisničkim zahtjevima. Na temelju tih zahtjeva, potrebno je odabrati prikladno sklopovlje i softver kako bi se zadovoljili specifični zahtjevi. Prije nego što se ugradbeni sustav stavi u produkciju, nužno je detaljno razmotriti plan budućnosti, uključujući i trajanje održavanja proizvoda. Primjerice, IoT komponenta za automobile može se masovno proizvoditi i integrirati u različite modele automobila, što olakšava generalizaciju procesa razvoja. S druge strane, kada je riječ o poljoprivrednim proizvodima za podršku, svaki sustav se mora individualno planirati i specificirati kako bi se prilagodio svakom poljoprivredniku i njegovom polju. U grubo bi sam razvojni proces mogli podijeliti na [4]:

1. **Generiranje ideje:** Identifikacija potrebe ili problema u određenom području koji bi se mogao riješiti pomoću IoT tehnologije. Nakon toga slijedi proces brainstorminga i istraživanja mogućih rješenja, te definiranje ciljeva i svrhe projekta.
2. **Analiza zahtjeva:** Ovdje se identificiraju ključne funkcionalnosti koje sustav treba imati kako bi zadovoljio korisničke potrebe. Razmatraju se tehnički zahtjevi, performanse, sigurnost, interoperabilnost i skalabilnost sustava. Također se definiraju korisnički slučajevi i scenariji korištenja.
3. **Analiza dionika:** Identificiraju se svi dionici koji su uključeni u projekt, poput korisnika, razvojnog tima i dobavljača. Razmatraju se njihove uloge, interakcije i očekivanja, te se definiraju potrebne integracije s vanjskim sustavima ili servisima.
4. **Definicija plana:** Ovaj korak uključuje planiranje rasporeda aktivnosti, resursa i rokova. Određuje se budžet i alokacija resursa, te se identificiraju rizici i strategije za njihovo upravljanje.
5. **Dizajn aplikacijske arhitekture:** Definira se arhitektura softverskih komponenti sustava, razrađuju se komunikacijski protokoli i sučelja, te se projektira korisničko sučelje i korisničko iskustvo.
6. **Dizajn platformne arhitekture:** Odabiru se potrebne hardverske komponente i senzori, razmatraju se sklopovske integracije i komunikacijski protokoli, te se planira infrastruktura za pohranu podataka i obradu u oblaku.
7. **Implementacija:** U ovom koraku provodi se razvoj softverskih aplikacija i algoritama za obradu podataka, integracija hardverskih komponenti i senzora, te testiranje jedinica, integracija i sustava.
8. **Testiranje:** Provjerava se funkcionalnost, performanse i sigurnost sustava, testira se kompatibilnost s različitim uređajima i platformama, te se provode simulacije stvarnih scenarija korištenja.
9. **Instalacija:** Slijedi implementacija sustava u stvarnom okruženju korisnika, postavljanje i konfiguracija hardverskih komponenti, te pružanje obuke i podrške korisnicima.

10. **Održavanje i podrška:** Na kraju, prati se performanse sustava i rješavaju eventualni problemi, ažurira se softver i firmware, te pruža tehnička podrška i održavanje korisničkih sustava.

Važno je napomenuti da navedeni koraci ne moraju nužno biti izvedeni u ovom redoslijedu, već ovise o samom projektu. Na nekim se koracima i detaljima može provesti više vremena i pažnje, dok se na drugima može manje. Ovaj opisani proces predstavlja okvir koji se može prilagoditi prema potrebama i zahtjevima konkretnog projekta, osiguravajući efikasno postizanje željenih rezultata.

4. Razvojna infrastruktura za ugradbene sustave

Prije nego što se usporede različite razvojne okoline za Internet stvari, važno je razumjeti osnovne elemente od kojih se sastoji jedna slika operacijskog sustava (*eng. image*) ciljana na ugradbene sustave. Operacijski sustav koji se najčešće pojavljuje u literaturi i industriji su derivacije Linuxa. Linux se često koristi zbog svoje otvorenosti i prisutnosti jakih zajednica koje doprinose različitim distribucijama. Ovo je prednost pred komercijalnim proizvodima jer omogućuje programerima razvoja da sami konfiguriraju svaki aspekt operacijskog sustava. Primjer korištenja Linuxa u modernim uređajima je Android, koji je također derivacija Linuxa. Čak su i Microsoftovi serveri postavljeni na Linuxu radi stabilnosti koju ovaj operacijski sustav pruža. Važno je napomenuti da se derivacije Linuxa obično ne koriste u strogim realno-vremenskim operativnim sustavima . Umjesto toga, za takve primjene često se koriste posebno dizajnirani operativni sustavi kao što su FreeRTOS, QNX, VxWorks i drugi, koji su optimizirani za zahtjeve s kritičnom vremenskom komponentom. RTOS su detaljnije razrađeni u poglavlju 4.3.1.

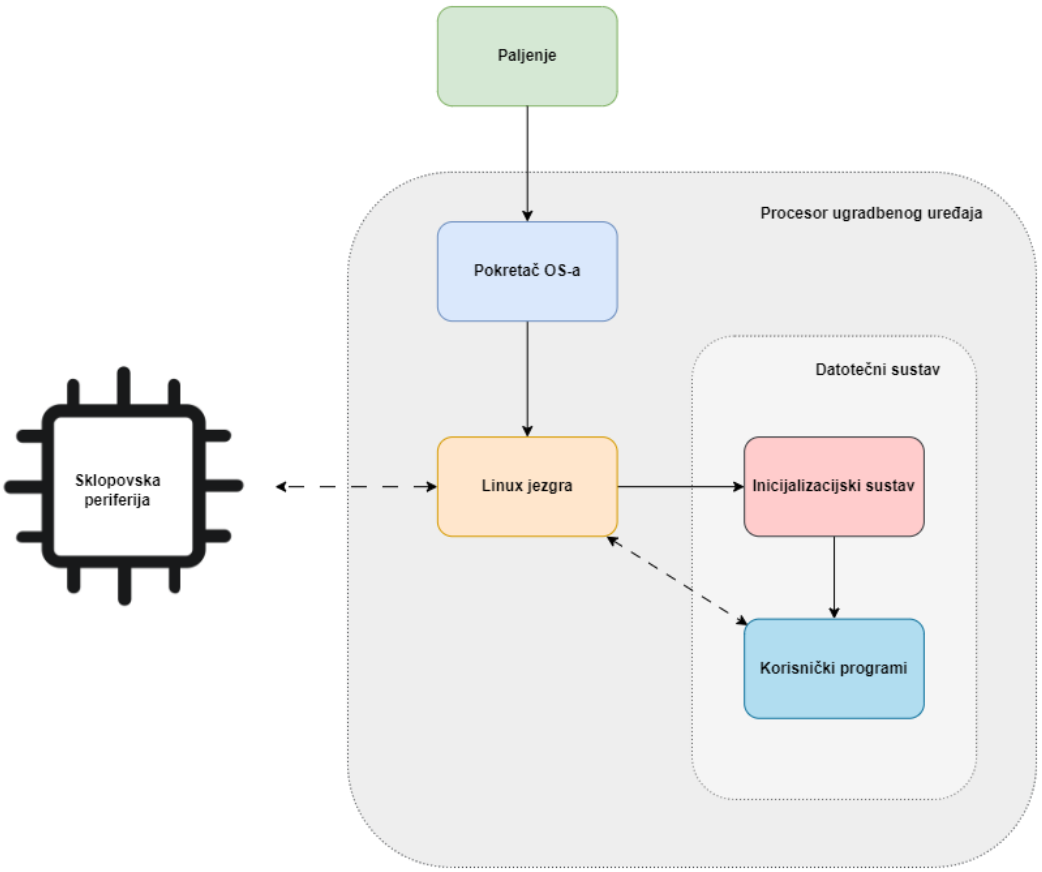
4.1. Osnovni elementi razvojne infrastrukture operacijskog sustava linux

Slika 3 prikazuje jednostavnu arhitekturu Linuxa za ugradbene uređaje. Vidljivo je da glavne komponente uključuju pokretač operacijskog sustava (*eng. bootloader*), jezgru Linuxovog operacijskog sustava (*eng. kernel*), datotečni sustav (*eng. file system*), koji se dalje sastoji od inicijalizacijskog sustava (*eng. init*), te prostora za korisničke programe.

4.1.1. Bootloader i Linux Kernel / Pokretački program, BIOS i jezgra

Kada se uređaj uključi, prvo se pokreće BIOS (*eng. Basic Input/Output System*) ili UEFI (*eng. Unified Extensible Firmware Interface*), ovisno o arhitekturi i konfiguraciji sustava. BIOS/UEFI izvršava početnu postavku hardvera i inicijalizira osnovne komponente, kao što su CPU, RAM i periferni uređaji. Proces pokretanja i inicijalizacije pokretanja prati sekcije prikazane na slici 3. Nakon što je BIOS/UEFI završio inicijalizaciju, slijedi faza u kojoj se učitava bootloader, kao što je GRUB (*eng. Grand Unified Bootloader*) ili LILO (*eng. Linux Loader*). Bootloader je odgovoran za pronalaženje i pokretanje binarnog programa operacijskog sustava. Uz pomoć bootloadera, sustav pronalazi odgovarajuću jezgru Linuxa na disku i učitava je u memoriju kako bi započela izvršavanje operacijskog sustava [6]. Među ugradbenim uređajima interneta stvari najpopularniji pokretački program je U-BOOT Bootloader. Ovaj pokretački program je otvorenog kôda te podržava brojne različite arhitekture procesora od ARM, x86 ili čak FPGA bazirane platforme, također dolazi i sa sučeljem komandne linije što razvojnim programerima omogućuje prilagodbe, manipulaciju particijama i cijelim procesom podizanja operacijskog sustava.

Uspješno učitan bootloader u memoriju pokreće svoj kôd, on pretražuje disk kako bi pronašao jezgru Linuxa. Kada pronađe jezgru, učitava je u memoriju i pokreće. Jezgra zatim



Slika 3: Jednostavni prikaz arhitekture linux ugradbenih uređaja [5]

započinje proces inicijalizacije hardvera, koji uključuje identifikaciju i konfiguraciju svih sklopovskih komponenti. Ovo uključuje i inicijalizaciju procesora, postavljanje osnovnih sustavnih registara, konfiguraciju memorije i upravljanje perifernim uređajima poput serijskih i paralelnih priključaka, USB uređaja, grafičkih kartica i drugih. Poslije inicijalizacije hardvera, jezgra nastavlja sa stvaranjem procesa i pokretanjem upravljačkih programa za datotečni sustav. Ovi programi su odgovorni za upravljanje datotekama i direktorijima, omogućujući korisničkim programima pristup podacima na disku. Jezgra također pokreće inicijalizacijski sustav, koji ima ulogu pokretanja korisničkih programa i usluga nakon što je sustav u potpunosti pokrenut. Inicijalizacijski sustav obično pokreće skripte i postavlja okruženje za korisničke programe, kao i pokreće razne servise i procese potrebne za normalan rad operativnog sustava. Kada su svi korisnički programi i servisi pokrenuti, sustav je spreman za interakciju s korisnikom ili s drugim uređajima na mreži.

Važno je istaknuti ključnu ulogu Linux jezgre i bootloadera u procesu pokretanja i inicijalizacije operativnog sustava na ugradbenim uređajima. Linux jezgra pruža osnovne funkcionalnosti za upravljanje hardverom i omogućuje izvršavanje korisničkih programa, dok bootloader osigurava uspješno pokretanje jezgre. Kroz njihovu suradnju, omogućena je stabilnost, pouzdanost i fleksibilnost u operativnom sustavu ugrađenim uređajima. Osim toga, razumijevanje ovih komponenti bitno je za dizajn i razvoj stabilnih i efikasnih sustava, te za osiguravanje njihove kompatibilnosti s različitim hardverskim platformama.

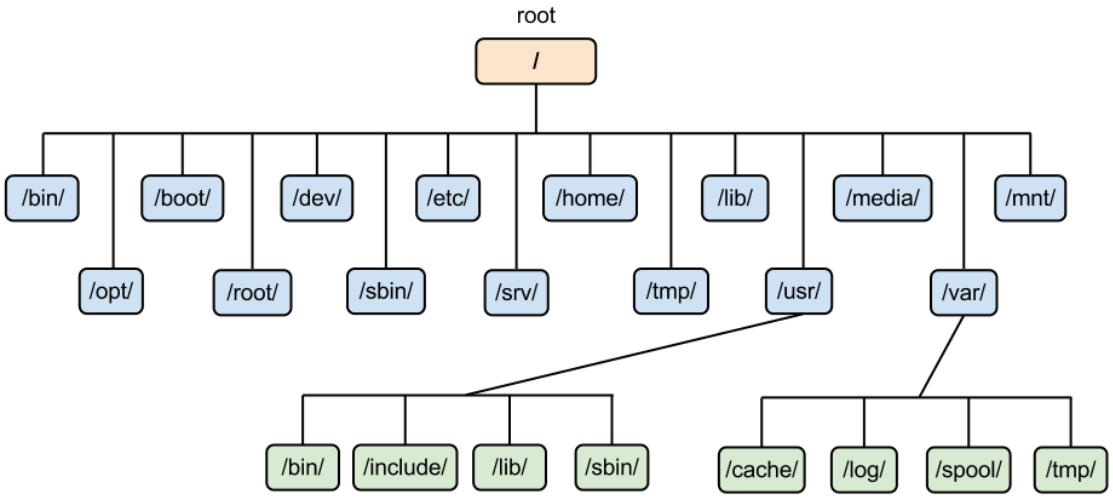
4.1.2. Datotečni sustav

Važna komponenta u Linux sustavima je korijenski datotečni sustav, poznat i kao *rootfs*. On predstavlja temeljni direktorij u sustavu datoteka koji sadrži sve osnovne datoteke potrebne za pokretanje Linux sustava, uključujući knjižnice i izvršne datoteke. Uobičajeno je da se bitne knjižnice nalaze unutar direktorija poput `/lib`, dok se ostale neosnovne knjižnice mogu smjestiti u direktorij `/usr/lib`. Osim knjižnica, u korijenskom datotečnom sustavu mogu se naći i osnovni programi poput naredbenog ljuske (*shell*) ili osnovnih binarnih datoteka, kao što su naredbe za montiranje (`mount`), te druge neosnovne binarne datoteke. BusyBox je softverski paket koji pruža jednu binarnu datoteku koja sadrži različite osnovne aplikacije. Te aplikacije obuhvaćaju korisne alate koji su ključni za ispravno funkcioniranje Linux operacijskog sustava. Važno je napomenuti da struktura korijenskog datotečnog sustava može varirati ovisno o konfiguraciji i zahtjevima projekta. Svaki ugradbeni uređaj može imati prilagođenu strukturu korijenskog datotečnog sustava prema svojim potrebama i specifičnim zahtjevima.

Na slici 4 prikazana je vizualizacija strukture korijenskog datotečnog sustava u Linuxu. Ova struktura pruža osnovni pregled ključnih direktorija i datoteka koje se mogu očekivati u korijenskom datotečnom sustavu. Važno je napomenuti da slika 4 služi kao ilustracija i da se stvarna struktura korijenskog datotečnog sustava može razlikovati ovisno o specifičnim zahtjevima i konfiguraciji uređaja. Tablica 2 proširuje sliku 4 opisujući svrhu određenih direktorija

Tablica 2: Opis direktorija Linuxa [8]

Direktorij	Opis
/	Korijenski datotečni sustav je najviši direktorij u sustavu datoteka. On mora sadržavati sve datoteke potrebne za pokretanje Linux sustava prije montiranja ostalih datotečnih sustava. U njemu moraju biti uključene sve izvršne datoteke i knjižnice potrebne za pokretanje preostalih datotečnih sustava. Nakon pokretanja sustava, svi ostali datotečni sustavi montiraju se na standardne i dobro definirane točke montiranja kao poddirektoriji korijenskog datotečnog sustava.
/bin	Direktorij /bin sadrži izvršne datoteke korisnika.
/boot	Sadrži statički bootloader i izvršnu datoteku jezgre, kao i konfiguracijske datoteke potrebne za pokretanje Linux računala.
/dev	Ovaj direktorij sadrži datoteke uređaja za svaki hardverski uređaj povezan na sustav. Ovo nisu upravljački programi uređaja, već datoteke koje predstavljaju svaki uređaj na računalu i olakšavaju pristup tim uređajima.
/etc	Sadrži lokalne konfiguracijske datoteke sustava za domaćin računala.
/home	Pohrana kućnih direktorija za korisničke datoteke. Svaki korisnik ima poddirektorij u /home.
/lib	Sadrži zajedničke knjižnice potrebne za pokretanje sustava.
/media	Mjesto za montiranje vanjskih izmjenjivih medijskih uređaja poput USB memorijskih pogona koji mogu biti povezani na domaćin.
/mnt	Privremena točka montiranja za redovite datotečne sustave (ne izmjenjive medije) koji se mogu koristiti dok administrator popravlja ili radi na datotečnom sustavu.
/opt	Ovdje se trebaju nalaziti opcionalne datoteke poput aplikacija koje su isporučili dobavljači.
/root	Ovo nije korijenski (/) datotečni sustav. To je kućni direktorij za korisnika root.
/sbin	Binarne datoteke sustava. To su izvršne datoteke koje se koriste za administraciju sustava.
/tmp	Privremeni direktorij. Koristi ga operativni sustav i mnogi programi za pohranu privremenih datoteka. Korisnici također mogu privremeno pohranjivati datoteke ovdje. Napomena: datoteke pohranjene ovdje mogu biti izbrisane u bilo kojem trenutku bez prethodne najave.
/usr	Ovo su dijeljene, samo za čitanje datoteke, uključujući izvršne datoteke i knjižnice, man datoteke i druge vrste dokumentacije.
/var	Ovdje se pohranjuju varijabilni podatkovni datoteke. To može uključivati datoteke dnevnika, MySQL i druge datoteke baze podataka, datoteke web poslužitelja, poštanske sandučiće i još mnogo toga.



Slika 4: Struktura Linux korijenskog datotečnog sustava [7]

4.1.3. Upravitelj inicijalizacije

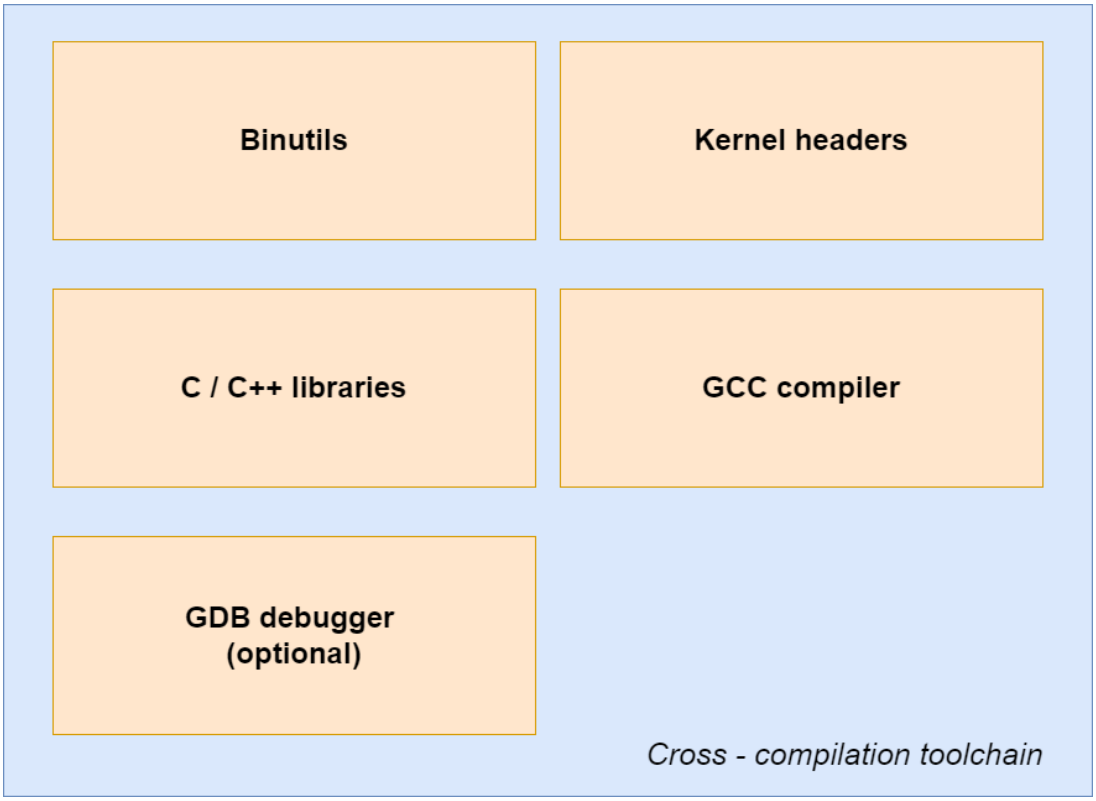
Upravitelj inicijalizacije (*eng. Init prmanager*) ima svrhu pokretanja servisa tijekom pokretanja sustava. Ti servisi mogu zahtijevati višestruke ovisnosti, a većina su pozadinski procesi (*eng. daemoni*). Jedni od najvažnijih inicijalizacijskih sustava i upravitelja servisa u Linuxu su systemd i SysVinit. Oni pružaju sustav dnevnika i alate potrebne za obavljanje administrativnih zadataka na sustavu. Servisi u kontekstu Linuxa su programi ili procesi koji se pokreću i rade u pozadini kako bi pružili određenu funkcionalnost ili uslugu sustavu ili korisnicima. Obično se automatski pokreću tijekom pokretanja sustava i mogu imati različite ovisnosti o drugim servisima ili sustavskim resursima. Servisi mogu obavljati različite zadatke, kao što su upravljanje mrežnim povezivanjem, pokretanje web poslužitelja, vođenje dnevnika sustava, upravljanje datotečnim sustavima itd. Također imaju ulogu praćenja definiranih ovisnosti između servisa i osiguravanja da se servisi pokreću u ispravnom redoslijedu kako bi se zadovoljile potrebne ovisnosti. Omogućuju sustavnicima da konfiguriraju pokretanje servisa i provjere status servisa kako bi osigurali da sustav pravilno funkcionira [9].

4.1.4. Upravitelj uređaja

Upravitelj uređaja (*eng. device manager*) igra ključnu ulogu u upravljanju hardverskim resursima računalnog sustava. Njegova osnovna zadaća je omogućiti operativnom sustavu pristup hardverskim uređajima kao što su diskovi, tipkovnice, miševi, mrežne kartice i drugi periferni uređaji. Kroz postupak prepoznavanja uređaja i montiranja, upravitelj uređaja osigurava da ti uređaji budu pravilno integrirani u operativni sustav i da budu dostupni korisnicima i aplikacijama koje ih trebaju koristiti. Osim toga, upravitelj uređaja može pružiti dodatne funkcionalnosti kao što su upravljanje energetskim postavkama uređaja, rukovanje uređajima u stanju mirovanja ili hibernacije, te rješavanje konflikata između različitih uređaja. Također, omogućuje praćenje statusa uređaja, detekciju grešaka i upravljanje njihovim ispravljanjem.

4.1.5. Cross-compiling alati / Alati križne kompilacije

Cross-compiling(*eng. Cross-compiling Toolchain* alati omogućuju razvoj softvera na jednom računalu (npr. osobnom računalu) za izvršavanje na drugom (npr. ugrađenom uređaju) s različitom arhitekturom procesora. Oni obično uključuju kompilatore, povezivače i druge potrebne alate prilagođene ciljanoj arhitekturi procesora. Ti alati su ključni za razvoj softvera za ugrađene sustave ili uređaje s ograničenim resursima, gdje se često koriste različite arhitekture procesora od onih na razvojnim računalima. Cross-compiling alati omogućuju programerima da efikasno razvijaju i testiraju softver za različite ciljane platforme bez potrebe za fizičkim prisustvom tih platformi. Pored osnovnih alata za kompilaciju, cross-compiling alati često uključuju i posebne SDK-ove (*eng. Software Development Kits*) koji sadrže dodatne biblioteke, zaglavlja i alate potrebne za razvoj softvera za određenu platformu ili uređaj. SDK-ovi omogućuju programerima da razvijaju aplikacije prilagođene specifičnim potrebama ciljane platforme, olakšavajući integraciju s postojećim sustavima i dodatnim funkcijama. Također, cross-compiling alati omogućuju programerima da efikasno koriste resurse razvojnog računala za razvoj i testiranje softvera, dok softver može biti optimiziran i prilagođen ciljanoj platformi prije nego što se prenese na nju. Slika 5 prikazuje primjer sadržaja jednog seta alata za križnu kompilaciju.



Slika 5: Sadržaj Cross-compiling alata [10]

Uz to, cross-compiling alati pružaju mogućnost emulacije ciljane platforme na razvojnom računalu, omogućujući programerima da testiraju i debagiraju svoj softver u kontroliranom okruženju prije nego što ga implementiraju na stvarnoj platformi. Ovi alati su ključni za brz i efikasan razvoj softvera za ugrađene sustave, omogućujući programerima da iskoriste prednosti modernog razvojnog procesa i alata, bez potrebe za fizičkim pristupom ciljanoj platformi

tijekom cijelog procesa razvoja.

4.1.6. Upravljanje procesom izgradnje

Sustavi za upravljanje procesom izgradnje olakšavaju automatizaciju procesa kompilacije, povezivanja i pakiranja softvera za ugradbene sustave. Oni pružaju okruženje i alate koji omogućuju programerima učinkovito upravljanje cijelim procesom razvoja, od izvornog kôda do gotovog proizvoda. Ovi sustavi obično uključuju skripte, konfiguracijske datoteke i druge resurse koji opisuju proces izgradnje softvera, uključujući korake kompilacije, povezivanja, generiranja paketa i druge potrebne operacije. Oni omogućuju programerima da jednostavno reproduciraju proces izgradnje i osiguraju dosljednost u izgradnji softvera na različitim okruženjima i platformama.

Jedan od najčešće korištenih alata za upravljanje procesom izgradnje u Linux okruženju je *Make*, koji koristi Makefile datoteke za definiranje procesa izgradnje. *Make* omogućuje programerima da definiraju pravila za kompilaciju i povezivanje izvornog kôda, te automatski generira ciljane izvršne datoteke ili pakete softvera. Osim toga, postoje i drugi alati poput *CMake*, *Autotools*, *Gradle* i drugi, koji pružaju naprednije mogućnosti za upravljanje procesom izgradnje, uključujući podršku za različite programerske jezike, platforme i složenije scenarije izgradnje. Upravljanje procesom izgradnje igra ključnu ulogu u razvoju softvera za ugrađene sustave, omogućujući programerima da učinkovito upravljaju kompleksnošću i veličinom projekata, osiguravajući kvalitetu i dosljednost u svim fazama razvoja.

4.2. Konceptualni razvojni okviri za ugradbene uređaje

Prije no što se usporede konkretne razvojne okoline, važno je razumjeti različite konceptualne pristupe koji su potrebni za razvoj različitih vrsta ugradbenih sustava, odnosno IoT Uređaja. Jedan od ključnih aspekata je potreba za adaptivnim sustavima, koji se ugrađuju u dinamično okruženje te moraju prilagoditi svoje ponašanje prema unutarnjim i vanjskim promjenama. Ovo zahtijeva promišljen pristup projektu i strukturiranje procesa razvoja kako bi se omogućila fleksibilnost i prilagodljivost ugrađenih sustava.

Razvoj adaptivnih sustava može biti izazovan jer se promjene događaju autonomno i nepredvidljivo, što može dovesti do fragmentacije i ad hoc pristupa u procesu razvoja. Zbog toga je važno primijeniti drugačiji sistematski pristup dizajnu, verifikaciji i validaciji kako bi se osigurala njihova stabilnost, pouzdanost i sigurnost, posebno u sigurnosno kritičnim primjenama. Osim toga, razumijevanje različitih konceptualnih okvira za razvoj ugradbenih sustava bitno je za uspješno vođenje projekata i ostvarivanje postavljenih ciljeva.

4.2.1. Modeliranje temeljno na modelima

MBSE (*eng. Model Based Software Engineering)* je proces razvoja softvera koji se bavi sve većom složenošću softverskog razvoja korištenjem apstrakcije i automatizacije [11]. Ap-

strakcija se postiže korištenjem odgovarajućih modela (dijelova) softverskog sustava, čime se olakšava razumijevanje složenosti sustava. Automatizacija sustavno pretvara te modele u izvršivi izvorni kôd, čime se ubrzava proces razvoja i smanjuje mogućnost ljudske greške. Ovo pruža učinkovit način za planiranje, dizajniranje i implementaciju ugradbenih sustava, omogućavajući razvojnicima da se usredotoče na funkcionalnosti sustava umjesto na tehničke detalje implementacije. Modeli se obično koriste kao sredstvo komunikacije između različitih dionika u procesu razvoja softvera, olakšavajući razmjenu ideja i koncepta te poboljšavajući suradnju među timovima. Ovo je od iznimne važnosti za kompliciranije projekte, ali ima i svojih nedostataka.

Neki od ključnih nedostataka MBSE su složenost procesa modeliranja, potreba za stručnošću i obukom u korištenju modeliranja alata te izazovi integracije modela u stvarne softverske proizvode, posebno u većim i složenijim sustavima. Iako MBSE obećava smanjenje kompleksnosti i skraćenje vremena razvoja softvera, postoje i izazovi koji proizlaze iz automatizacije procesa. Na primjer, iako su alati za podršku MBSE-u omogućili brz i jednostavan razvoj generatora kôda za proizvoljne modele, ova automatizacija može postati neefikasna kada su generatori kôda složeniji. Kod generiranih sistema, često se suočavamo s potrebom prilagođavanja generatora kôda, što zahtijeva dodatne napore u razvoju i održavanju sustava. Osim toga, zahtjevi za prilagodbom modela različitim domenama ili scenarijima također mogu dovesti do izazova u dizajniranju i razvoju specifičnih domenskih jezika (*eng. Domain Specific Languages*). Unatoč tim izazovima, korištenje MBSE-a može biti korisno u situacijama gdje je naglasak na implementaciji nove arhitekture softvera, iako praksa MBSE-a nije uvijek optimalna kada je riječ o automatizaciji procesa razvoja softvera [12].

4.2.2. Modeliranje temeljno na komponentama/komponente

CBSE (*eng. Component-Based Software Engineering*) je pristup razvoju softvera koji se fokusira na ponovnu uporabu komponenti umjesto na kreiranje softvera od nule [13]. U ovom pristupu, softverski sustavi se konstruiraju spajanjem postojećih komponenti, što omogućuje brži razvoj softvera uz poboljšanu kvalitetu i skalabilnost. Za razliku od MBSE-a, CBSE pruža manju razinu apstrakcije, što ga čini relevantnijim u kontekstu razvoja ugradbenih sustava interneta stvari.

Primarni cilj CBSE-a je povećati produktivnost, smanjiti troškove i poboljšati kvalitetu softverskih proizvoda. Korištenjem postojećih komponenata, razvojni timovi mogu ubrzati proces razvoja, izbjeći ponovno izmišljanje kotača te se usredotočiti na dodavanje vrijednosti kroz prilagodbu i integraciju komponenti. Ovaj pristup pruža fleksibilnost i skalabilnost u razvoju softverskih sustava, što ga čini atraktivnim za različite domene primjene. Međutim, kao i svaki pristup razvoju softvera, CBSE također nosi određene izazove i nedostatke. Jedan od ključnih izazova je pronalaženje odgovarajućih komponenti za specifične zahtjeve projekta. Također, potrebno je pažljivo razmatrati arhitekturu sustava kako bi se osiguralo da komponente međusobno djeluju na očekivani način. Osim toga, integracija komponenti može biti kompleksna, posebno u većim i složenijim sustavima, što zahtijeva pažljivo planiranje i testiranje. Unatoč tim izazovima, CBSE ostaje koristan pristup za razvoj softverskih sustava, posebno u situacijama

gdje je naglasak na ponovnoj uporabi komponenti i brzini razvoja softvera [13].

4.2.3. Rubus model komponenti

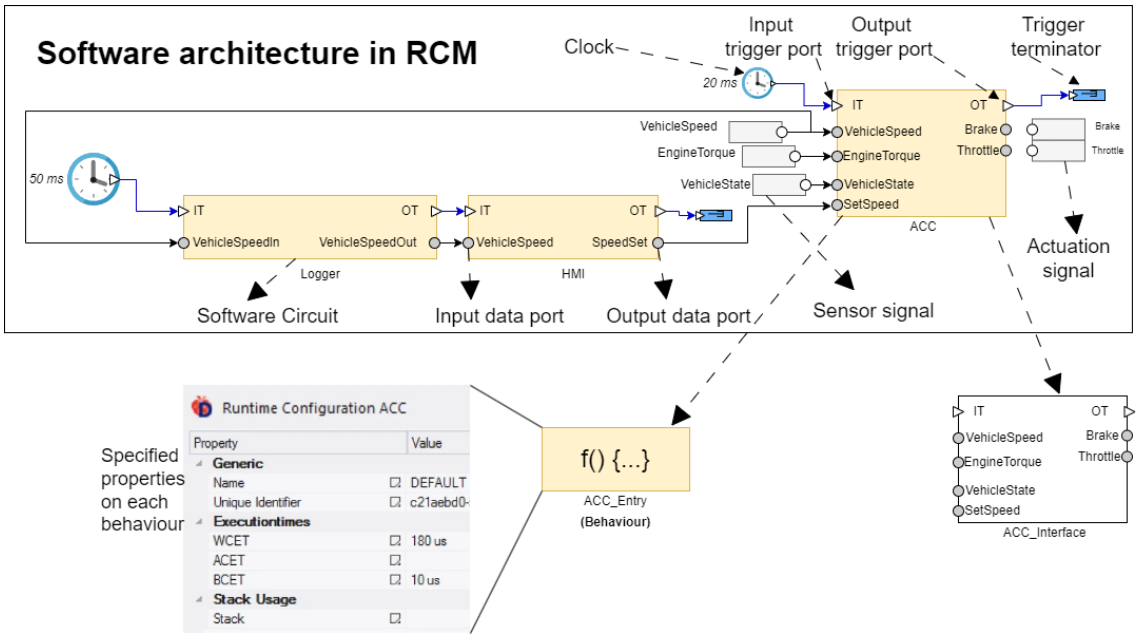
Rubus model komponenti (*eng. Rubus component model - RCM*), predstavlja domenski specifični jezik za razvoj kontrolnih funkcija u resursno ograničenim, realno-vremenskim ugradbenim sustavima, često korišten u automobilskoj industriji radi razvoja sigurnosno-kritičnih softverskih sustava [14]. Ovaj model omogućuje detaljno modeliranje komponenti i njihovih međusobnih interakcija, pružajući programerima i inženjerima alate za precizno definiranje ponašanja softvera u raznim situacijama. Kroz RCM, mogu se specificirati zahtjevi vezani uz vrijeme izvršavanja, alokaciju resursa i povezanost komponenti unutar sustava. Osim u automobilskoj industriji, RCM se također može primijeniti u drugim domenama gdje su potrebni visoko predvidljivi i mjerljivi ugradbeni sustavi, poput industrijske automatizacije ili medicinskih uređaja.

Iako pruža napredne mogućnosti za razvoj sigurnosno-kritičnih softverskih sustava, RCM također nosi određene nedostatke i izazove. Jedan od nedostataka je kompleksnost samog okvira, što može otežati učenje i usvajanje za novopridošle inženjere. Složenost integracije s postojećim sustavima također može predstavljati izazov, jer RCM zahtijeva prilagodbu i usklađivanje s postojećim arhitekturama i tehnologijama. Također, budući da se RCM često koristi za razvoj softvera u sigurnosno-kritičnim okruženjima, certifikacijski procesi mogu dodatno produžiti vrijeme razvoja i povećati troškove implementacije. Unatoč tim izazovima, RCM ostaje važan alat u razvoju sofisticiranih ugradbenih sustava koji zahtijevaju visoku razinu pouzdanosti i performansi.

Slika 6 prikazuje primjer softverske arhitekture modela u RCM-u. Na slici se može vidjeti vizualni prikaz softverskog kruga (*eng. software circuit*) koji predstavlja jedinicu funkcionalnosti u RCM-u. Softverski krug sadrži ulazne podatkovne portove (*eng. software circuit input data port*) i izlazne podatkovne portove (*eng. software circuit output data port*), koji omogućuju prijenos podataka između različitih komponenti u sustavu. Također se na slici nalaze signali senzora (*eng. sensor signals*), sat (*eng. clock*), ulazni okidački portovi (*eng. input trigger port*), izlazni okidački portovi (*eng. output trigger port*), terminator okidača (*eng. trigger terminator*) te signali aktivacije (*eng. actuation signals*). Uz to, slika sadrži komponente poput AC Interface i ACC Entry, koje predstavljaju specifična ponašanja unutar softverskog kruga. Svako ponašanje ima svoje posebne karakteristike i zadatke, što omogućuje programerima da precizno definiraju funkcionalnosti softvera u raznim situacijama. Ovaj primjer ilustrira kako se kroz RCM mogu modelirati složene arhitekture softvera, pružajući programerima alate za precizno definiranje ponašanja softvera u realno-vremenskim ugradbenim sustavima.

4.2.4. TEACHING

TEACHING predstavlja distribuiranu i pouzdanu umjetnu inteligenciju (*eng. Artificial Intelligence - AI*) koja integrira kontinuiranu ljudsku povratnu informaciju, podržavajući dizajn i implementaciju aplikacija za CPSoS (*eng. Cyber-Physical Systems of Systems*). Ovo je projekt predstavljen u sklopu TEACHING Projekta, financiran od strane EU Horizona za istraživanje i



Slika 6: Primjer softverske arhitekture modela u RCM [14]

inovacije, koji omogućuje održivo ljudsko sudjelovanje u razvoju, optimizaciji i nadzoru inteligentnih autonomnih sustava s rigoroznim dokazima ekvivalencije i pouzdanosti. TEACHING prihvaća viziju čovjeka u središtu autonomnog CPSoS-a, naglašavajući ljudski-centrirani dizajn i formalnu validaciju koja prelazi paradigme. Korištenjem specijaliziranih AI računalnih struktura i simulacija za inteligentna sigurnosna rješenja, TEACHING razvija CPS (*eng. Cyber-Physical System*) usmjeren na čovjeka za autonomne sigurnosno-kritične aplikacije, temeljene na distribuiranoj, energetski učinkovitoj i pouzdanoj umjetnoj inteligenciji [15].

Slika 7 prikazuje slojeve TEACHING platforme.

- **Sloj apstrakcije infrastrukture (IAL)** (*eng. Infrastructure Abstraction Layer*): IAL pruža jedinstveni sloj apstrakcije za izvođenje aplikacija (kôda ili komponenti). U osnovi, on homogenizira underlying infrastrukture pružajući jedan API za implementaciju, izvođenje i praćenje resursa i komponenti aplikacija.
- **Okruženje izvršavanja/upravljanja (EME)** (*eng. Execution/Management Environment*): EME izlaže jedinstveni API koji olakšava izvršavanje i upravljanje životnim ciklusom komponenti aplikacija. Pruža runtime i integrirane knjižnice koje pružaju usluge i optimizacije na gornjim slojevima.
- **TEACHING alatni set za programiranje (SDK)** (*eng. TEACHING Software Toolkit*): TEACHING SDK pruža okvir za implementaciju aplikacija CPS-a. Pruža API-je (*eng. Application Processing Interface*) za implementaciju aplikacija koje mogu raditi na TEACHING platformi koristeći najbolje usluge CPS-a. Ovaj alatni set podržava šest različitih alatki:
    1. **AI toolkit**: Omogućuje razvoj aplikacija temeljenih na umjetnoj inteligenciji, uključujući alate za učenje, treniranje i evaluaciju modela.

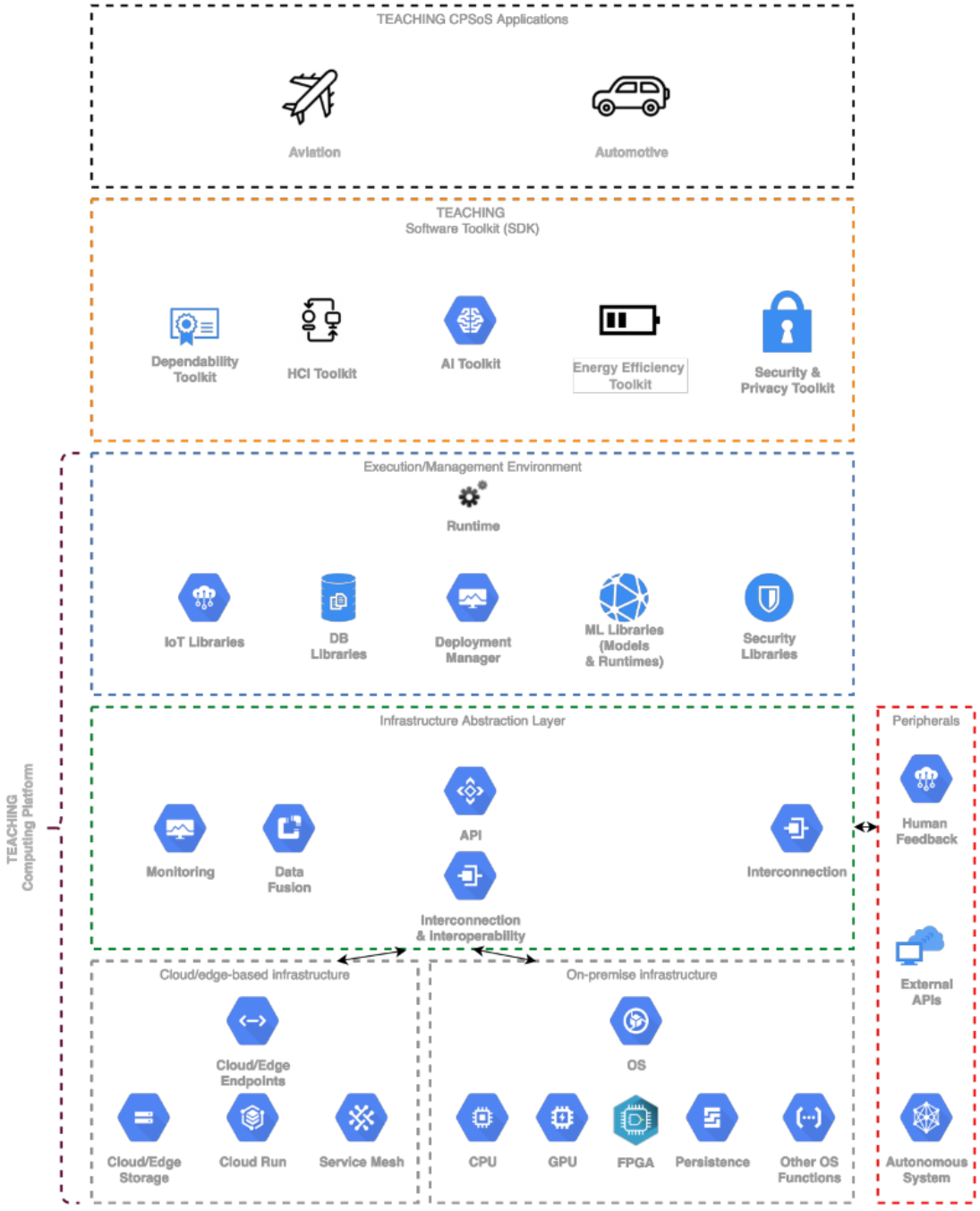
2. **HCI toolkit**: Pruža alate za integraciju ljudske interakcije u aplikacije, uključujući filtre i mehanizme za upravljanje ljudskim povratnim informacijama.
3. **Security and Privacy toolkit**: Sadrži alate za osiguranje sigurnosti i privatnosti aplikacija, uključujući sigurnosne API-je i smjernice za zaštitu podataka.
4. **Dependability toolkit**: Pruža alate za osiguranje pouzdanosti i stabilnosti aplikacija, uključujući audit kôda i implementaciju inženjerskih obrazaca.
5. **Energy Efficiency toolkit**: Omogućuje optimizaciju energetske učinkovitosti aplikacija kroz upravljanje resursima poput dinamičkog upravljanja naponom i frekvencijom.
6. **TEACHING CPSoS Applications**: Podržava razvoj i izvršavanje aplikacija za CPSoS, uključujući nezavisne komponente ili cijele aplikacije koje se izvršavaju unutar TEACHING SDK-a.

TEACHING predstavlja ključnu tehnološku inovaciju koja omogućuje integraciju naprednih AI tehnologija u konceptualne razvojne okvire ugradbenih uređaja. Kroz svoju distribuiranu i pouzdanu umjetnu inteligenciju, TEACHING pruža temelje za razvoj autonomnih, prilagodljivih i sigurnosno-kritičnih sustava, što predstavlja suštinski korak naprijed u postizanju ciljeva konceptualnih razvojnih okvira ugradbenih uređaja.

4.3. Razvojne okoline ugradbenih sustava

Nakon što su razjašnjeni konceptualni okviri, pristupi i specifični elementi koji čine ugradbene sustave interneta stvari, sada je vrijeme da se usporede različite okoline za razvoj ugradbenih sustava. U ovom pregledu, razvojne okoline će se podijeliti na operacijske sustave u realnom vremenu(*eng. Real-Time Operating Systems - RTOS*) te operacijske sustave za generalnu namjenu (*eng. General Purpose Operating System - GPOS*) Iako se nazivaju operacijskim sustavima, u ovom kontekstu ti sustavi predstavljaju razvojne okoline jer jezgrena komponenta koju pružaju važna za izgradnju ugradbenih sustava. Operacijski sustav u kontekstu ugradbenih sustava interneta stvari ima ključnu ulogu jer upravlja svim resursima i funkcionalnostima uređaja, osigurava upravljanje hardverom, raspoređivanje resursa, upravljanje memorijom, podršku za mrežne komunikacije te pruža sučelje za izvršavanje aplikacija i ostale već prijašnje spomenutih elemenata infrastrukture.

Operacijski sustavi za ugradbene sustave interneta stvari obično su dizajnirani da budu lagani, brzi i energetski učinkoviti te pružaju podršku za rad u stvarnom vremenu, što je važno za aplikacije koje zahtijevaju brze i točne odgovore, poput sustava za praćenje i upravljanje senzorima ili sustava za nadzor okolišnih uvjeta. Također, operacijski sustavi ugradbenih uređaja često imaju posebne značajke za sigurnost i pouzdanost, s obzirom na to da se ti uređaji koriste u kritičnim okruženjima poput već spomenutih medicinskih uređaja ili industrijskih kontrolnih sustava.



Slika 7: TEACHING arhitekture [15]

U tablici 3 prikazane su ključne razlike između ove dvije kategorije operacijskih sustava po različitim aspektima, a ovi operacijski sustavi bit će dodatno istraženi u sljedećim potpoglavljima.

4.3.1. Operacijski sustavi u realnom vremenu

U poglavlju ugradbenih sustava također su ugradbene sustavi podijeljeni na ugradbene sustave u realnom vremenu i opće ugradbene sustave. Ovo potpoglavlje namijenjeno je da dublje istraži te sustave ali kroz pogleda operacijskog sustava.

Tablica 3: Usporedba GPOS i RTOS za ugradbene uređaje

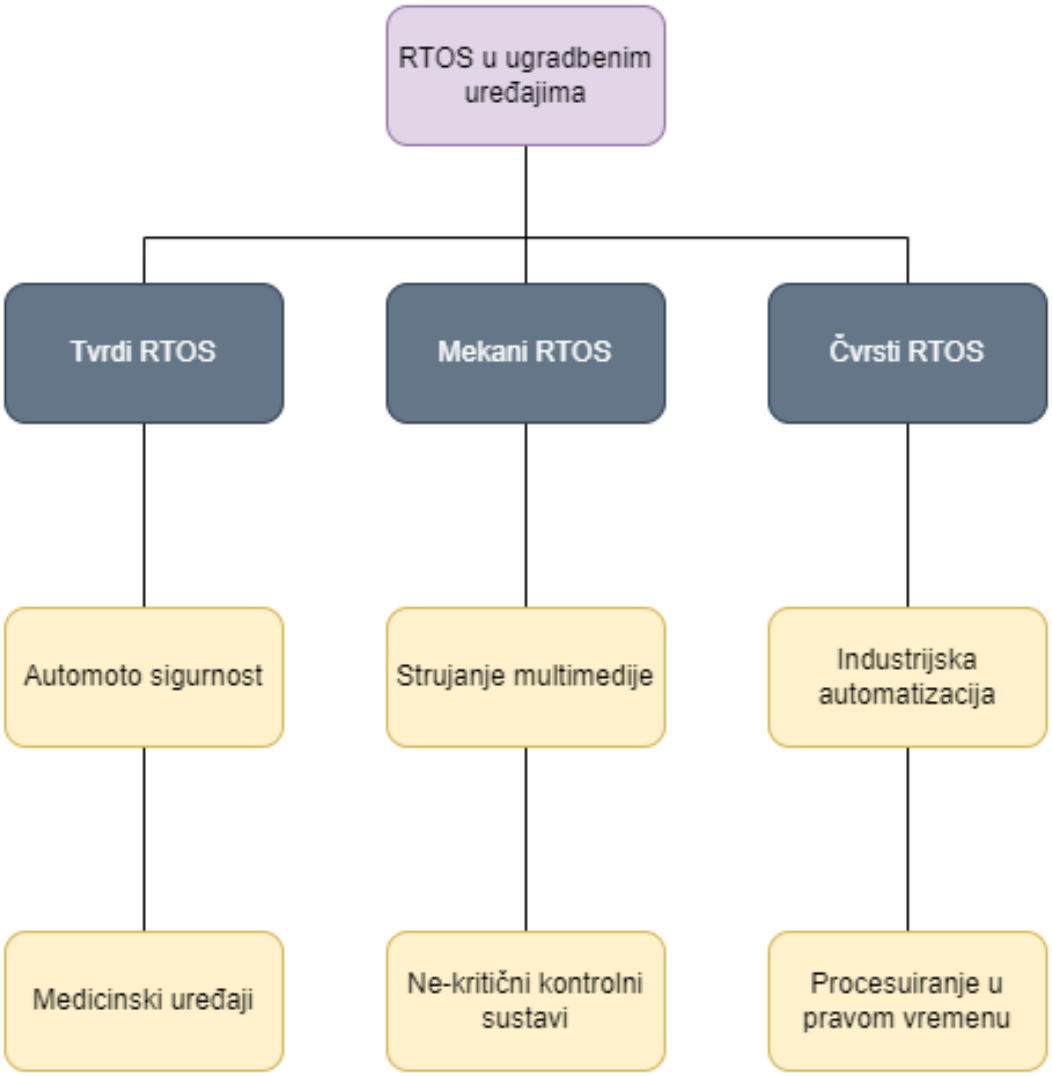
Aspekt	**Opći operacijski sustav**	**RTOS**
Prikladnost za aplikacije u stvarnom vremenu	Nije prikladan za aplikacije u stvarnom vremenu zbog varijabilne latencije	Prilagođen je zahtjevima stvarnog vremena, osiguravajući precizno vrijeme i nisku latenciju
Fokus i prioritet	Balansira višezadaćnost i fleksibilnost, što ga čini pogodnim za širok spektar aplikacija	Prioritizira preciznost i izvedbu u stvarnom vremenu nad višezadaćnošću, što je najbolje za zadatke koji zahtijevaju vremenski kritičnost
Izbor aplikacija	Tipično se koristi u širokom spektru računalnih potreba, nudeći fleksibilnost	Preferira se za ugradbene sustave sa strožim zahtjevima za vrijeme i zadatke u stvarnom vremenu
Kriteriji odabira	Odabir bi trebao biti temeljen na specifičnim potrebama i ograničenjima vremena ugradbenog sustava	Odabire se kada je determinističko i predvidivo vrijeme od presudne važnosti, a izvedba u stvarnom vremenu je prioritet

Neka osnovna podjela ovakvih sustava prikazana je na slici 8. Operacijski sustavi u pravom vremenu su specijalizirani operacijski sustavi dizajnirani za ugradbene sustave. Oni osiguravaju da se zadaci izvršavaju unutar određenih vremenskih ograničenja, čineći ih vitalnom komponentom za aplikacije koje zahtijevaju izvedbu u stvarnom vremenu. RTOS-ovi dolaze u različitim vrstama, uključujući [16]:

- Tvrdi operacijski sustavi u stvarnom vremenu: U tvrdi operacijskim sustavima u stvarnom vremenu, ispunjavanje rokova je strogi zahtjev. Neuspjeh u završetku zadatka unutar njegovog određenog vremenskog okvira može imati ozbiljne posljedice. Primjeri takvih sustava uključuju aktivaciju zračnih jastuka u automobilima i pejsmejkera u zdravstvu.
- Mekani operacijski sustavi u stvarnom vremenu: Mekani operacijski sustavi u stvarnom vremenu mogu tolerirati povremena kašnjenja u rokovima. Ako zadatak povremeno ne uspije ispuniti svoj rok, to možda neće rezultirati značajnom štetom ili kvarom sustava. Takva tolerancija na kašnjenja omogućava bolje iskorištavanje resursa i fleksibilnost u rasporedu zadataka. Primjeri aplikacija koje mogu koristiti mekane operacijske sustave u stvarnom vremenu su reproduktori streaming medija i sustavi za video konferencije.
- Čvrsti operacijski sustavi u stvarnom vremenu: Čvrsti operacijski sustavi u stvarnom vremenu zahtijevaju ispunjavanje rokova. Iako se povremena kašnjenja mogu

prihvatiti, često propuštanje rokova može dovesti do nestabilnosti sustava ili čak kritičnih kvarova. Čvrsti RTOS osigurava da se zadaci izvršavaju na vrijeme uz visoku pouzdanost.

Ova klasifikacija omogućuje odabir odgovarajućeg tipa operacijskog sustava u stvarnom vremenu ovisno o zahtjevima aplikacije i toleranciji na kašnjenja u zadacima. Slika 8 pruža vizualni prikaz ove podjele.



Slika 8: Podjela pravovremenih operacijskih sustava

Tablica 4 prikazuje različite popularne pravovremene operacijske sustave, njihov opis, primjenu i tip. *Real-time* aplikacije postaju sve popularnije zbog sve veće složenosti u sustavima. Razvojni inženjeri se često suočavaju s izazovom odabira odgovarajućeg real-time operacijskog sustava (RTOS) koji će zadovoljiti zahtjeve aplikacije u smislu troškova, pouzdanosti, brzine i energetske učinkovitosti. Postoje mnoge opcije dostupne na tržištu, uključujući komercijalne RTOS-ove i otvorene kôdne RTOS-ove.

Primjeri popularnih RTOS-ova uključuju *FreeRTOS*, *Zephyr*, *RIOT*, *QNX* i *VxWorks*.

FreeRTOS je popularan otvoreni RTOS koji se često koristi u malim i resursno ograničenim ugradbenim sustavima. Zephyr, također otvoreni RTOS, optimiziran je za IoT uređaje i pametne naprave. RIOT je fokusiran na podršku za IoT uređaje i mrežnu komunikaciju, dok su QNX i VxWorks komercijalni RTOS-ovi koji nude visoku pouzdanost i skalabilnost. Ovi RTOS-ovi imaju široku primjenu u različitim industrijama, uključujući IoT uređaje, ugradbene sustave, automobilsku industriju, medicinske uređaje, telekomunikacije i avioniku. Odabir između komercijalnih i otvorenih kôdnih RTOS-ova ovisi o potrebama projekta, budžetu i preferencijama razvojnog tima [17].

4.3.2. Opći operacijski sustavi

Za razliku od RTOS-a koji su usmjereni na specifične vremenske zahtjeve, opći operacijski sustavi (*General Purpose Operating System - GPOS*) pružaju širu aplikativnu podršku, ali su i dalje od velike važnosti za ugradbene sustave. Ovi sustavi, poput Embedded Linuxa, imaju široku primjenu u različitim industrijskim kontekstima, uključujući kontrolu i dijagnostiku industrijskih strojeva. Primjer takvog sustava je Programabilna kontrolna i komunikacijska platforma (eng. *Programmable Control and Communication Platform - PCCP*), koja se koristi za kontrolu i dijagnostiku industrijskih strojeva. Ovi sustavi se razlikuju od klasičnih desktop sustava po tome što su osjetljivi na troškove, imaju vremenska ograničenja i zahtjeve u pogledu potrošnje energije. Operacijski sustav u PCCP-u je *Embedded Linux*, što omogućuje široku primjenu iste hardverske platforme s različitim softverskim rješenjima. Softverski stog temelji se na Linuxu s upravljačkim programima za komunikacijska sučelja i aplikacijama za kontrolu i dijagnostiku strojeva. Aplikacije se izvode na Linuxu bez izravnog pristupa hardveru, što omogućuje izolaciju aplikacija od hardvera korištenjem posredničkog sloja nazvanog *Hardware-dependent Software - HdS*. Ovaj pristup omogućuje promjene u hardveru bez potrebe za izmjenama u aplikacijama. Kontrola strojeva u PCCP-u ostvarena je s pomoću CODESYS okvira, koji slijedi IEC 61131 standard za programiranje industrijskih kontrolnih sustava i aplikacija. CODESYS upravlja sučeljima sustava pružajući način za ostvarenje kontrole industrijskog stroja. Osim toga, web poslužitelj se koristi za pristup u svrhu dijagnostike [18].

Važna prednost općih operacijskih sustava poput u kontekstu Embedded Linuxa u usporedbi s RTOS-ovima je šira primjena i veća fleksibilnost u različitim industrijama i aplikacijama. Dok RTOS-ovi nude pravovremeno izvršavanje zadataka, opći operacijski sustavi omogućuju bolje razdvajanje aplikacija i hardvera, što olakšava održavanje i prilagodbu sustava. Osim toga, opći operacijski sustavi imaju bogatije razvojne alate i podršku za šire područje hardverskih platformi, što ih čini pogodnijima za dugoročne projekte s promjenjivim hardverom i zahtjevima.

U tablici 5 prikazana je usporedba različitih razvojnih okvira ugradbenih sustava. Dva popularna okvira koja se ističu su *Buildroot* i *Yocto Project*. Oba su otvorenog kôda i podržavaju širok spektar arhitektura, uključujući *ARM, PPC, MIPS, x86, i x86-64*. *OpenEmbedded* je također važan projekt u području ugradbenog razvoja. Kao alat za izgradnju, OpenEmbedded koristi BitBake, koji kontrolira kako se izgrađuju stvari i ovisnosti o izgradnji. Za razliku od

Tablica 4: Usporedba različitih pravovremenih operacijskih sustava

Operacijski sustav	**Opis**	**Primjena**	**Tip**
FreeRTOS	Popularan otvoreni OS, jednostavan za korištenje s minimalnom jezgrom	IoT uređaji, ugradbeni sustavi	Otvoreni kôd
Zephyr	Otvoreni OS optimiziran za IoT uređaje s bogatim skupom značajki	IoT uređaji, pametne naprave	Otvoreni kôd
RIOT	Otvoreni OS fokusiran na podršku za IoT uređaje i mrežnu komunikaciju	IoT uređaji, senzorske mreže	Otvoreni kôd
QNX	Komercijalni OS s visokom pouzdanošću i deterministčkim ponašanjem	Automobilska industrija, medicinski uređaji	Komercijalan
VxWorks	Komercijalni OS s visokom pouzdanošću i skalabilnošću	Telekomunikacije, avionika	Komercijalan

Tablica 5: Usporedba različitih razvojnih okvira ugradbenih sustava [18]

Okvir	Podržane arhitekture	Osnovan	Vrsta projekta	Dokumen-tacija	Usluge podrške	Ažuriranja	Zajednice	Partnerske organizacije
LTIB	ARM,PPC, Coldifre	2005	Otvoreni kôd	+	Freescale Inc.	Neaktivna	++	Freescale Inc.
Buildroot	ARM,PPC, MIPS, x86	1999	Otvoreni kôd	++	Nema	Svakih 6 mjeseci	+++	-
Open-Embedded	ARM,PPC, MIPS, x86, x86-64, amd64, avr64	2004	Otvoreni kôd	+++	Nezavisni konzultanti	Regularno	+++	-
Yocto project	ARM,PPC, MIPS, x86, x86-64	2010	Otvoreni kôd	+++	Nezavisni konzultanti	Svakih 6 mjeseci	+++	Intel, Huawei, Texas Instruments, Wind River, MontaVisa
TimeSys LinuxLink PRO	ARM,PPC, MIPS, INTEL Atmel	1995	Komercijalno	+++	Da	Regularno	+	-
MontaVisa Linux	ARM,PPC, MIPS, x86, x86-64, amd64	1999	Komercijalni	+++	da	Regularno	+	-
Wind River Linux	ARM, PPC, MIPS, Intel	1981	Komercijalni	+++	Da	Mjesečno	+	Freescale, ARM, Intel, Texas Instruments, MIPS

pojedinačnih alata za izgradnju poput make-a, BitBake ne temelji se na jednom makefile-u ili zatvorenom skupu međusobno ovisnih makefile-ova, već prikuplja i upravlja otvorenim skupom uglavnom neovisnih opisa izgradnje (recepta za pakiranje) i gradi ih u pravilnom redoslijedu [19]. *OpenEmbedded* se koristi za križno kompiliranje, pakiranje i instaliranje softverskih paketa te je korišten za izgradnju i održavanje nekoliko ugradbenih Linux distribucija, uključujući OpenZaurus, Ångström, Familiar i SlugOS. Primarna upotreba OpenEmbeddeda uključuje rukovanje križnim kompiliranjem, rukovanje ovisnostima između paketa, stvaranje paketa i slika te podršku za različite arhitekture, distribucije i strojeve.

Buildroot je stariji od *Yocto Projecta* i pruža jednostavno i brzo rješenje za izgradnju ugradbenih sustava. Omogućuje korisnicima da generiraju prilagođene Linux distribucije prilagođene njihovim specifičnim potrebama. Buildroot se temelji na ideji izgradnje izvora s nula konfiguracija. To znači da korisnici definiraju konfiguracijske datoteke koje specificiraju koje pakete žele uključiti u svoju distribuciju, te Buildroot automatski preuzima, kompilira i integrira te pakete. Ova jednostavnost korištenja i konfiguracije čini Buildroot pogodnim za manje projekte i početnike u području ugradbenog razvoja.

S druge strane, *Yocto Project* je napredniji i fleksibilniji okvir koji omogućuje detaljnu kontrolu nad procesom izgradnje. Yocto Project proširuje OpenEmbedded projekt, pružajući korisnicima alate, metapodatke i okruženje potrebno za izradu prilagođenih Linux distribucija [19]. Ovo proširenje omogućava Yocto Projectu da bude napredniji i fleksibilniji okvir od *Buildroota*, uz pružanje detaljne kontrole nad procesom izgradnje. Jedna od glavnih prednosti Yocto Projecta je njegova sposobnost upravljanja složenim arhitekturama i hardverom te podrška za različite procesorske arhitekture. Također, Yocto Project nudi bogatu dokumentaciju, podršku za širok spektar platformi i česta ažuriranja, što olakšava održavanje i razvoj ugradbenih sustava. Osim toga, korisnici mogu prilagoditi Yocto Project svojim specifičnim zahtjevima projekta, što ga čini popularnim okvirom u industriji, posebno za kompleksnije ugradbene projekte gdje je potrebna visoka razina kontrole i prilagodbe. Komercijalni embedded softverski okviri, poput proizvoda tvrtki *Wind River* i *MontaVista*, također se temelje na Yocto Projectu. Yocto Project također operira pod okriljem Linux Foundationa.[18].

Okviri otvorenog kôda imaju snažnu podršku zajednice i veliku bazu korisnika koji aktivno doprinose razvoju i održavanju. Ova otvorenost i podrška zajednice daje im prednost nad komercijalnim projektima jer omogućuje korisnicima prilagodbu, poboljšanja i dijeljenje resursa i znanja unutar zajednice ugradbenih razvojnih inženjera.

5. Detaljna usporedba dviju odabranih razvojnih okolina

Kako bi se dobio bolji uvid u razliku između procesa razvoja raznih okolina odabrane su dvije koje će pokazati neke fundamentalne razlike. Dvije odabrane razvoje okoline su Yocto Project i Buildroot. Cilj ovog poglavlja je pružiti dublji uvid u korake aspekte korištenja ovih razvojnih okvira te prikazati njihove razlike i prednosti u stvarnim scenarijima razvoja ugradbenih sustava.

5.1. Analiza razvojnog procesa Buildroot-a

Buildroot je alat koji pojednostavljuje i automatizira proces izgradnje kompletnog Linux sustava za ugradbeni sustav koristeći križnu kompilaciju. Kako bi to postigao Buildroot može generirati sve već spomenute dijelove arhitekture linuxa poput korijenskog datotečnog sustava, sliku Linux jezgre, pokretač OS-a te alatni lanac križne kompilacije za ciljni sustav. Buildroot se može koristiti za bilo koju kombinaciju ovih opcijama, na primjer: moguće je koristiti postojeći alatni lanac za križnu kompilaciju i izgraditi samo svoj korijenski datotečni sustav s Buildrootom. Buildroot ima ciklus izdavanja novih inačica otprilike jednom u tromjesečnom periodu. Podržava brojne procesore i njihove varijante poput ARM, MIPS i sl; također dolazi s osnovnim konfiguracijama za nekoliko gotovih ploča dostupnih na tržištu. Osim toga, brojni projekti trećih strana temelje se na Buildrootu ili razvijaju svoje pakete za podršku ploča (*eng. Board Support Package - BSP*) ili alatni lanac za razvoj softvera (*eng. Software Development Kit - SDK* na osnovi Buildroota [10].

Sam razvojni proces operacijskih sustava je sustavno jednostavan. Nakon preuzimanja Buildroota sa službenih stranica (Slika 9) postoji nekoliko opcija. Da bi se brzo postavila virtualna mašina s potrebnim ovisnostima, može se koristiti Vagrantfile iz direktorija *support/misc/Vagrantfile* u Buildroot izvornoj stablu. Za postavljanje izoliranog Buildroot okruženja na Linuxu ili Mac OS X-u, potrebno je unijeti sljedeću naredbu u terminal:

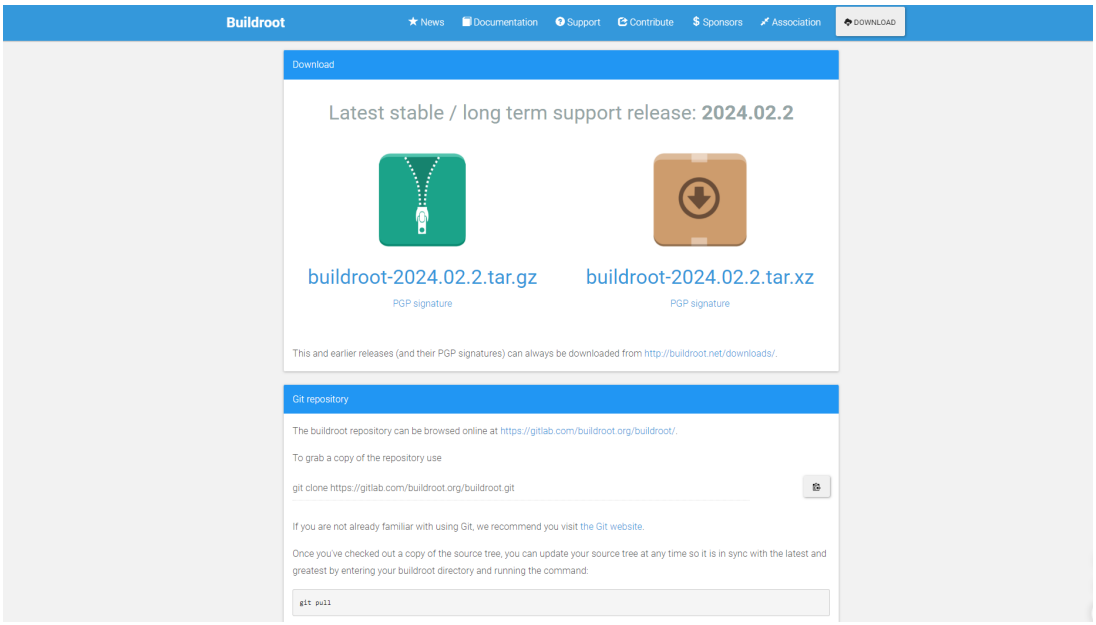
```
curl -O https://buildroot.org/downloads/Vagrantfile; vagrant up
```

Za korisnike Windowsa, naredba za PowerShell je:

```
(new-object System.Net.WebClient).DownloadFile(
"https://buildroot.org/downloads/Vagrantfile","Vagrantfile");
vagrant up
```

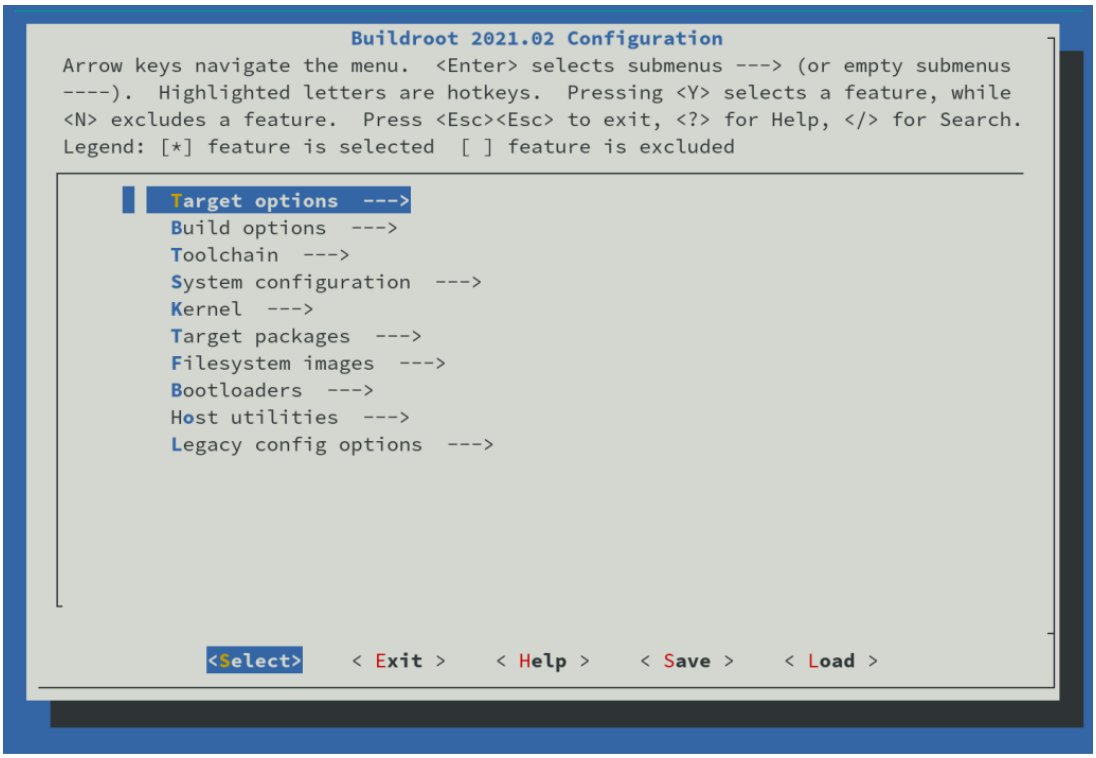
Prvi korak u korištenju Buildroota je kreiranje konfiguracije. Buildroot nudi alat za konfiguraciju sličan onome u Linux kernelu ili BusyBoxu, odnosno izbornik u terminalu koji izgledom podsjeća na BIOS (Slika 10). Iz Buildroot direktorija pokreće se jedna od sljedećih naredbi:

```
$ make menuconfig
```



Slika 9: Službena stranica preuzimanja [20]

```
$ make nconfig
$ make xconfig
$ make gconfig
```



Slika 10: Izbornik `menuconfig` [10]

Svaki konfigurator je temeljen na drugoj tehnologiji. Svaka od ovih naredbi će izgraditi alat za konfiguraciju, pa je moguće da će biti potrebno instalirati "razvojne" pakete za relevantne biblioteke koje koriste alati za konfiguraciju. Za svaku stavku u alatu za konfiguraciju može se

pronaći pomoć koja opisuje svrhu stavke. Nakon što je sve konfigurirano, alat za konfiguraciju generira .config datoteku koja sadrži cijelu konfiguraciju. Ovu datoteku čita top-level Makefile. Kako bi se pokrenuo cijeli proces izgradnje jednostavno se u komandnoj liniji pozove sljedeća naredba:

```
$ make
```

Prema osnovnim zadanim postavkama, Buildroot ne podržava top-level paralelnu izgradnju, pa pokretanje `make -jN` nije potrebno, iako trenutno postoje eksperimentalna verzija paralelne izgradnje [10]. Naredba `make` će obično izvršiti sljedeće korake:

- preuzimanje izvornih datoteka (po potrebi);
- konfiguracija, izgradnja i instalacija alata za cross-kompilaciju, ili jednostavno uvoz vanjskog alata za kompilaciju;
- konfiguracija, izgradnja i instalacija odabranih ciljnih paketa;
- izgradnja kernel slike, ako je odabrana;
- izgradnja bootloader slike, ako je odabrana;
- kreiranje korijenskog datotečnog sustava u odabranim formatima.

Izlaz Buildroota pohranjen je u jedinstvenom direktoriju, `output/`. Ovaj direktorij sadrži nekoliko poddirektorija:

- `images/` gdje su pohranjene sve slike (kernel slika, bootloader i slike root datotečnog sustava). Ovo su datoteke koje je potrebno staviti na ciljni sustav.
- `build/` gdje su izgrađene sve komponente (uključujući alate potrebne Buildrootu na hostu i pakete kompilirane za cilj). Ovaj direktorij sadrži jednu podmapu za svaku od ovih komponenti.
- `host/` sadrži alate izgrađene za host, i `sysroot` ciljnog alata za kompilaciju. Prvi je instalacija alata kompiliranih za host koji su potrebni za pravilno izvršavanje Buildroota, uključujući alat za cross-kompilaciju. Drugi je hijerarhija slična hijerarhiji root datotečnog sustava. Sadrži zaglavlja i biblioteke svih korisničkih paketa koji pružaju i instaliraju biblioteke koje koriste drugi paketi. Međutim, ovaj direktorij nije namijenjen kao root datotečni sustav za cilj: sadrži puno razvojnih datoteka, nestripirane binarne datoteke i biblioteke koje ga čine prevelikim za ugrađeni sustav. Ove razvojne datoteke koriste se za kompilaciju biblioteka i aplikacija za cilj koje ovise o drugim bibliotekama.
- `staging/` je simbolička poveznica na `sysroot` ciljnog alata za kompilaciju unutar `host/`, koja postoji zbog kompatibilnosti unatrag.

- `target/` koji sadrži gotovo kompletan root datotečni sustav za cilj: sve potrebno je prisutno osim datoteka uređaja u `/dev/` (Buildroot ih ne može kreirati jer ne radi kao root i ne želi raditi kao root). Također, nema ispravne dozvole (npr. `setuid` za `busybox` binarnu datoteku). Stoga, ovaj direktorij ne bi trebao biti korišten na cilju. Umjesto toga, treba koristiti jednu od slika izgrađenih u direktoriju `images/`. Ako je potrebna izdvojena slika root datotečnog sustava za pokretanje preko NFS-a, tada treba koristiti `tarball` sliku generiranu u `images/` i izdvojiti je kao root. U usporedbi s `staging/`, `target/` sadrži samo datoteke i biblioteke potrebne za pokretanje odabranih ciljnih aplikacija: razvojne datoteke (zaglavlja, itd.) nisu prisutne, binarne datoteke su o.

Navedene naredbe, `make menuconfig|nconfig|gconfig|xconfig` i `make`, osnovne su koje omogućuju jednostavno i brzo generiranje slika koje odgovaraju potrebama korisnika, sa svim značajkama i aplikacijama koje su u procesu omogućene. Naredbe `make V=1` , `make list-defconfigs` i `make help` pružaju korisne informacije za učinkovitu upotrebu Buildroota, omogućujući prikaz svih izvršenih naredbi, popis dostupnih defconfig određenih ploča te pregled svih dostupnih ciljeva. Također, čišćenje je neophodno nakon promjene opcija konfiguracije arhitekture ili alata, a `make clean` služi za brisanje svih proizvoda izgradnje. Za generiranje priručnika, tu su naredbe `make manual-clean` i `make manual`, dok se za potpuno brisanje svih proizvoda i konfiguracije koristi `make distclean`. Osim toga, korisne su i naredbe za ispisivanje unutarnjih make varijabli i manipulaciju njima. Također, važno je razumjeti kada je potrebna potpuna ponovna izgradnja sustava, što ovisi o promjenama u konfiguraciji arhitekture ili alata, dodavanju ili uklanjanju paketa te promjenama u konfiguraciji datotečnog sustava. Za ponovnu izgradnju paketa, koriste se naredbe poput `make -rebuild`, `make -reconfigure` i `make -dirclean`. Nakon toga, Buildroot također podržava izgradnju izvan glavnog direktorija izvorne datoteke, što omogućuje korištenje različitih direktorija izlaza. Konačno, Buildroot poštuje neke okolišne varijable koje se koriste za upravljanje kompilacijom, kao što su `HOSTCXX`, `HOSTCC`, `UCLIBC_CONFIG_FILE` i `BUSYBOX_CONFIG_FILE`.

Dodatno, Buildroot omogućuje generiranje izvještaja i analizu rezultata izgradnje koristeći nekoliko alata i ciljeva. Ovi alati pružaju uvid u sastav generiranog sustava, potencijalne sigurnosne ranjivosti te ovisnosti između paketa. Primjerice, cilj `show-info` stvara JSON sažetak koji detaljno opisuje omogućene pakete, njihove ovisnosti, licence i ostale metapodatke. Ove informacije pomažu korisnicima da razumiju komponente prisutne u njihovoj konfiguraciji izgradnje.

Osim toga, cilj `pkg-stats` generira detaljne izvještaje o paketima u HTML i JSON formatima. Ti izvještaji uključuju informacije o poznatim CVE ranjivostima koje utječu na pakete, kao i dostupne nadogradnje za te pakete iz upstream izvora. Također, sposobnost Buildroota da generira grafove ovisnosti pomaže u vizualizaciji veza između paketa. Pokretanjem cilja `graph-depends`, korisnici mogu generirati graf koji prikazuje ovisnosti unutar kompiliranog sustava. Taj graf može biti koristan za razumijevanje uloge različitih komponenti u ugrađenom Linux sustavu. Nadalje, korisnici mogu prilagoditi ponašanje generiranja grafova ovisnosti postavljanjem opcija u varijablu okoline `BR2_GRAPH_DEPS_OPTS`. Te opcije omogućuju korisnicima kontrolu dubine ovisnosti, isključivanje određenih paketa te odabir boja koje se koriste u

prikazu grafa, među ostalim postavkama.

Buildroot također olakšava upotrebu vlastitih programa ili softvera koji nisu uključeni u Buildroot pakete. Korisnici mogu koristiti alatni lanac generiran od strane Buildroot-a, koji se nalazi u output/host/ direktoriju, dodajući output/host/bin/ u PATH okolinu. Također, Buildroot omogućava eksport alatnog lanca i razvojnih datoteka svih odabranih paketa kao SDK, koji se može distribuirati razvojnim programerima za razvoj njihovih aplikacija izvan Buildroot-a. Nakon ekstrakcije SDK tarball-a, korisnik mora pokrenuti skriptu relocate-sdk.sh kako bi se osiguralo da su sve putanje ažurirane na novu lokaciju.

Slika 11 prikazuje tok izgradnje operacijskog sustava s pomoću buildroot-a. Sastoji se od sljedećih važnih koraka:

1. Preuzimanje paketa: Prvi korak je preuzimanje potrebnih izvornih paketa softvera koji će biti uključeni u izgradnju operacijskog sustava. Ovi paketi mogu uključivati različite komponente, poput jezgre Linuxa, biblioteka, alata i aplikacija.

2. Konfiguracija alatnog lanca: Nakon preuzimanja paketa, slijedi konfiguracija alatnog lanca ili skupa alata koji će se koristiti za izgradnju softvera. Ovo uključuje postavljanje parametara poput verzije kompajlera, alata za povezivanje i ostalih alata potrebnih za proces izgradnje.

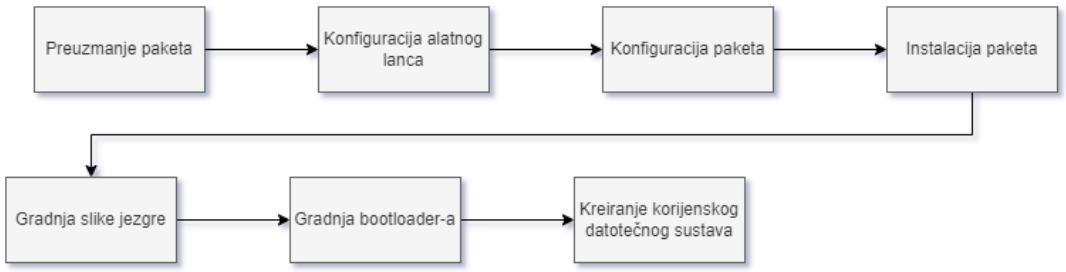
3. Konfiguracija paketa: Sljedeći korak je konfiguracija paketa, što podrazumijeva postavljanje opcija i parametara za svaki pojedini softverski paket koji će biti uključen u konačni operacijski sustav. Ovdje se specificiraju značajke i opcije koje će biti prisutne ili isključene u konačnoj instalaciji.

4. Instalacija paketa: Nakon konfiguracije, slijedi instalacija paketa, što uključuje proces kompilacije i povezivanja izvornog koda kako bi se generirali izvršni programi, biblioteke i ostale komponente koje čine operacijski sustav.

5. Gradnja slike jezgre: Jedan od važnih koraka je gradnja slike jezgre, što uključuje kompilaciju izvornog koda jezgre Linuxa i povezivanje u binarnu datoteku koja će biti učitana prilikom pokretanja operacijskog sustava.

6. Gradnja bootloader-a: Nakon izgradnje jezgre, slijedi gradnja bootloader-a, što uključuje kompilaciju bootloader-a i postavljanje parametara za pokretanje operacijskog sustava.

7. Kreiranje korijenskog datotečnog sustava: Konačni korak u procesu izgradnje je kreiranje korijenskog datotečnog sustava, koji uključuje pakiranje svih izgrađenih komponenti u sustavu, zajedno s konfiguracijskim datotekama i drugim potrebnim datotekama, u format koji će biti spreman za instalaciju i upotrebu na ciljnom uređaju.



Slika 11: Proces izgradnje buildroota, na temelju [10]

5.1.1. Preporučena struktura direktorija

Prilikom prilagođavanja Buildroot-a projektu, stvara se jedna ili više datoteka specifičnih za projekt koje treba negdje pohraniti. Iako većina ovih datoteka može biti smještena na bilo kojem mjestu jer se njihova putanja specificira u konfiguraciji Buildroot-a, razvojni programeri Buildroot-a preporučuju određenu strukturu direktorija koja je opisana u ovom odjeljku [10].

Određenja ovoj strukturi direktorija, možete odabrati gdje ćete smjestiti samu strukturu: ili unutar stabla Buildroot-a, ili izvan njega koristeći vanjsko stablo br2-external. Obe opcije su valjane, izbor je na korisniku.

```
+-- board/
|   +-- /
|       +-- /
|           +-- linux.config
|           +-- busybox.config
|           +--
|           +-- post_build.sh
|           +-- post_image.sh
|           +-- rootfs_overlay/
|           |   +-- etc/
|           |   +--
|           +-- patches/
|               +-- foo/
|               |   +--
|               +-- libbar/
|                   +--
|
+-- configs/
|   +-- _defconfig
|
+-- package/
|   +-- /
|       +-- Config.in (ako ne koristite br2-external stablo)
```

```
|       +-- .mk (ako ne koristite br2-external stablo)
|       +-- package1/
|       |    +-- Config.in
|       |    +-- package1.mk
|       +-- package2/
|           +-- Config.in
|           +-- package2.mk
|
+-- Config.in (ako koristite br2-external stablo)
+-- external.mk (ako koristite br2-external stablo)
+-- external.desc (ako koristite br2-external stablo)
```

Ako se ova struktura smjesti izvan stabla Buildroot-a, ali unutar br2-external stabla, komponente i potencijalno mogu biti suvisle te se mogu izostaviti

Vrlo je uobičajeno da korisnik ima nekoliko povezanih projekata koji djelomično zahtijevaju iste prilagodbe. Umjesto dupliciranja ovih prilagodbi za svaki projekt, preporučuje se korištenje slojevite prilagodbe. Gotovo sve metode prilagodbe dostupne u Buildroot-u, poput skripti za nakon izgradnje i preklapanja korijenskog datotečnog sustava, prihvaćaju popis stavki razdvojenih razmakom. Navedene stavke uvijek se obrađuju po redu, s lijeva na desno. Stvaranjem više takvih stavki, jednu za zajedničke prilagodbe i drugu za stvarno projektno specifične prilagodbe, moguće je izbjeći nepotrebno dupliciranje. Svaki sloj obično je predstavljen odvojenim direktorijem unutar board//.

Primjer strukture direktorija gdje korisnik ima dva sloja prilagodbi common i fooboard je [10]:

```
+-- board/
    +-- /
        +-- common/
        |   +-- post_build.sh
        |   +-- rootfs_overlay/
        |   |   +-- ...
        |   +-- patches/
        |       +-- ...
        |
        +-- fooboard/
            +-- linux.config
            +-- busybox.config
            +--
            +-- post_build.sh
            +-- rootfs_overlay/
            |   +-- ...
            +-- patches/
                +-- ...
```

5.1.2. Upravljanje konfiguracijskim datotekama i prilagodbe

Korisnici mogu prilagoditi svoje ciljne datotečne sustave u Buildrootu na nekoliko načina. Primjerice, postavljanjem opcije `BR2_GLOBAL_PATCH_DIR` na određenu vrijednost, Buildroot će primijeniti zakrpe iz više direktorija u zadanom redoslijedu. Na taj način prvo se primjenjuju zakrpe iz zajedničkog sloja, a zatim zakrpe iz specifičnog sloja za određenu ploču. Ovaj pristup omogućuje modularno i fleksibilno održavanje zakrpa za različite dijelove sustava, što olakšava upravljanje promjenama i poboljšanjima u različitim fazama razvoja i održavanja. Osim promjene konfiguracije putem naredbe `make *config`, postoje i drugi preporučeni načini prilagodbe ciljnih datotečnih sustava, kao što su root filesystem overlay-i i post-build skripte. Root filesystem overlay omogućava kopiranje stabla datoteka izravno preko ciljnog datotečnog sustava nakon njegove izgradnje. Ova značajka se omogućuje postavljanjem opcije `BR2_ROOTFS_OVERLAY`, pri čemu se može specificirati više overlay-eva odvojenih razmacima. S druge strane, post-build skripte se konfiguriraju putem opcije `BR2_ROOTFS_POST_BUILD_SCRIPT` i omogućuju izvršavanje skripti koje modificiraju ciljni datotečni sustav nakon izgradnje svih odabranih softverskih paketa, ali prije sastavljanja rootfs slika.

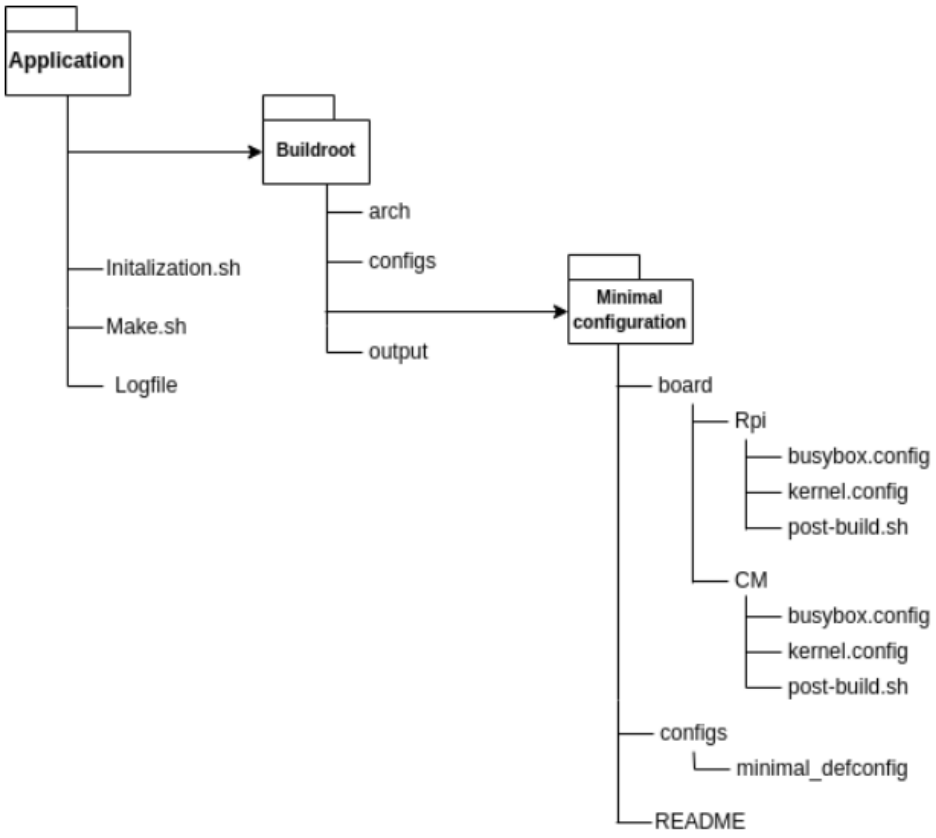
Uz root filesystem overlay-e i post-build skripte, Buildroot nudi i nekoliko drugih metoda za prilagodbu ciljnih datotečnih sustava, iako se ne preporučuju. Izravna modifikacija ciljnog datotečnog sustava omogućuje brze testove, ali promjene neće preživjeti naredbu `make clean`. Prilagodba putem vlastitog skeletona može biti korisna ako zadani skeleton ne zadovoljava specifične potrebe, no duplicira skeleton i ne koristi kasnije poboljšanja zadane verzije. Post-fakeroot skripte omogućuju modifikacije koje zahtijevaju root prava, ali se preporučuje korištenje postojećih mehanizama za postavljanje dozvola ili stvaranje korisničkih računa umjesto oslanjanja na fakeroot. Ove metode, iako korisne u određenim situacijama, zahtijevaju pažljivo razmatranje i planiranje kako bi se osigurala dugoročna održivost i kompatibilnost prilagođenih rješenja. Integracija različitih komponenti u sustav opisan je u ovom poglavlju. Buildroot je visoko konfigurabilan, što znači da gotovo sve što se ovdje spominje može biti izmijenjeno ili nadjačano rootfs overlayem ili prilagođenom skeleton konfiguracijom. To omogućava korisnicima da prilagode sustav svojim specifičnim potrebama, osiguravajući da različiti dijelovi sustava rade zajedno na optimalan način. Jedan od ključnih aspekata u Buildrootu je upravljanje sigurnosnim modulima poput SELinuxa, koji omogućuje provođenje detaljnih sigurnosnih politika unutar sustava. SELinux nudi tri načina rada: isključen, permisivni i obavezni. Kroz Buildroot korisnici mogu kontrolirati način rada SELinuxa te omogućiti podršku za njega odabirom odgovarajućih konfiguracijskih opcija. Ovo omogućava prilagodbu sigurnosnih postavki prema potrebama specifičnih aplikacija i okruženja.

Licenciranje je također bitan aspekt koji korisnici moraju uzeti u obzir. Korištenje open-source softvera dolazi s određenim obavezama, poput redistribucije izvornog koda i pridržavanja uvjeta licence. Buildroot pruža alate za prikupljanje svih potrebnih materijala za usklađivanje s licencama, no korisnici moraju biti svjesni da neki materijali možda neće biti automatski uključeni te je potrebno ručno ih prikupiti kako bi se ispunile sve pravne obveze. Korisnici mogu naići na razne probleme tijekom korištenja Buildroota, kao što su problemi s pokretanjem sus-

tava, nedostatak kompajlera ili razvojnih datoteka na ciljnom sustavu. Jednostavna rješenja uključuju podešavanje sistemskih postavki, korištenje vanjskih alata ili prebacivanje na prikladnije distribucije ukoliko Buildroot ne zadovoljava njihove potrebe. Ova fleksibilnost i mogućnost prilagodbe ključni su za uspješno upravljanje i održavanje ugrađenih sustava razvijenih s pomoću Buildroota [10].

5.1.3. Primjer konfiguracije s Raspberry Pi

Prilikom konfiguriranja Buildroot-a za upotrebu s Raspberry Pi platformom, korisnik će slijediti preporučenu strukturu direktorija kako bi organizirao svoj projekt. Slika 12 prikazuje konkretnu implementaciju radne strukture direktorija, utemeljena na prijašnjoj predloženoj strukturi.



Slika 12: Radni direktorij Raspberry Pi Buildroot projekta [21]

Na slici se može vidjeti struktura direktorija projekta. Glavni direktorij projekta sadrži poddirektorij "buildroot" koji sadrži potrebne datoteke za konfiguraciju i izgradnju sustava. Također, postoje direktoriji "`Initialization.sh`" i "`Make.sh`" koji služe za pokretanje inicijalizacije i izgradnje projekta, te "`Logfile`" koji bilježi izlazne podatke tijekom procesa izgradnje. Unutar "*buildroot*" direktorija nalaze se poddirektoriji "*arch*" i "*configs*", koji sadrže arhitekturne specifičnosti i konfiguracijske datoteke. Konkretni primjeri konfiguracijskih datoteka prikazani su unutar poddirektorija "*Minimal configuration*", gdje su datoteke podijeljene prema različitim pločama (u ovom slučaju Raspberry Pi i CM). Ove datoteke definiraju konfiguracijske pos-

tavke za operacijski sustav, pakete i post-build skripte potrebne za izgradnju prilagođenog OS-a. Pravilno organiziranje radnog direktorija olakšava upravljanje projektom i osigurava da sve potrebne datoteke i skripte budu na odgovarajućim mjestima, čime se olakšava prilagodba i izgradnja prilagođenog OS-a za Raspberry Pi platformu.

Kroz ovu analizu, detaljno su prikazani koraci potrebni za razvoj i prilagodbu sustava korištenjem Buildroot-a. Praktični aspekti, poput kreiranja konfiguracija, upravljanja zakrpama i strukturiranja direktorija, pokazuju fleksibilnost i prilagodljivost ovog alata u stvarnim razvojnim scenarijima. Međutim, postoji važna problematika održavanja koja se može pojaviti nakon nekoliko godina korištenja. S obzirom na dinamičnu prirodu ugradbenih tehnologija, određeni dijelovi sustava izgrađenog pomoću Buildroot-a mogu postati zastarjeli ili nepodržani, što može predstavljati izazove u ažuriranju na nove verzije komponenti ili sigurnosne zakrpe. Na primjer, ako je sustav izgrađen s pomoću Buildroot-a koristio određenu biblioteku koja je tada bila popularna i aktivno održavana. No, nakon nekoliko godina, ta se biblioteka može potencijalno zamijeniti novijom tehnologijom ili može prestati primati podršku i ažuriranja. U takvim slučajevima, integriranje novih verzija ili zamjena dotrajalih komponenti može biti izazovno, dovodeći do potrebe za dubljim prilagodbama ili čak restrukturiranjem cijelog sustava. Ovo nije izazov koji je specifičan samo za Buildroot, već je to opći problem održavanja ugradbenih i IoT uređaja. Ipak, neki alati mogu pružiti olakšice u rješavanju ovog problema, pružajući napredne mehanizme za upravljanje ažuriranjima i prilagodbama komponenti.

Kako bi se stekao cjelovitiji uvid u razvojne okoline za ugradbene sustave, potrebno je razmotriti i alternativne alate. U sljedećem dijelu, analizirat će se razvojni proces koristeći Yocto Project, uspoređujući njegove mogućnosti s Buildroot-om te ističući specifične prednosti i izazove koji se pojavljuju pri njegovoj upotrebi. Yocto nudi drugačiji pristup koji može biti izuzetno koristan u složenijim i prilagođenijim scenarijima razvoja ugradbenih sustava te jednostavniji za održavanje.

5.2. Analiza razvojnog procesa Yocto project-a

The Yocto Project je otvoreni kolaborativni projekt koji pomaže razvojnim inženjerima u stvaranju prilagođenih Linux-baziranih sustava za ugrađene proizvode i druge ciljane okoline, bez obzira na arhitekturu hardvera. Projekt pruža fleksibilni skup alata i prostor gdje ugrađeni razvojni inženjeri širom svijeta mogu dijeliti tehnologije, softverske sklopove, konfiguracije i najbolje prakse koje se mogu koristiti za stvaranje prilagođenih Linux i RTOS slika za ugrađene uređaje. Ovaj odjeljak analizira razvojni proces Yocto Project-a i ključne elemente koji ga čine, uključujući arhitekturu, alate i procese koji omogućuju izgradnju prilagođenih operacijskih sustava za ugrađene sustave. Proces izgradnje koristi standard za isporuku hardverske podrške i softverskih sklopova, omogućavajući razmjenu softverskih konfiguracija i izgradnji. Ovo može biti između dobavljača i potrošača hardvera ili omogućavanje suradnje u i oko softverskog ekosustava, na primjer: alati omogućuju korisnicima izgradnju i podršku prilagodbi za više platformi hardvera i softverskih sklopova na održiv i razmjeren način [22]. Projekt se također oslanja i surađuje s OpenEmbedded projektom, koji je sistem izgradnje koji se koristi i dijeli komponente s Yocto Project-om.

Yocto Project kombinira i održava nekoliko ključnih elemenata, kako je prikazano na slici 13:

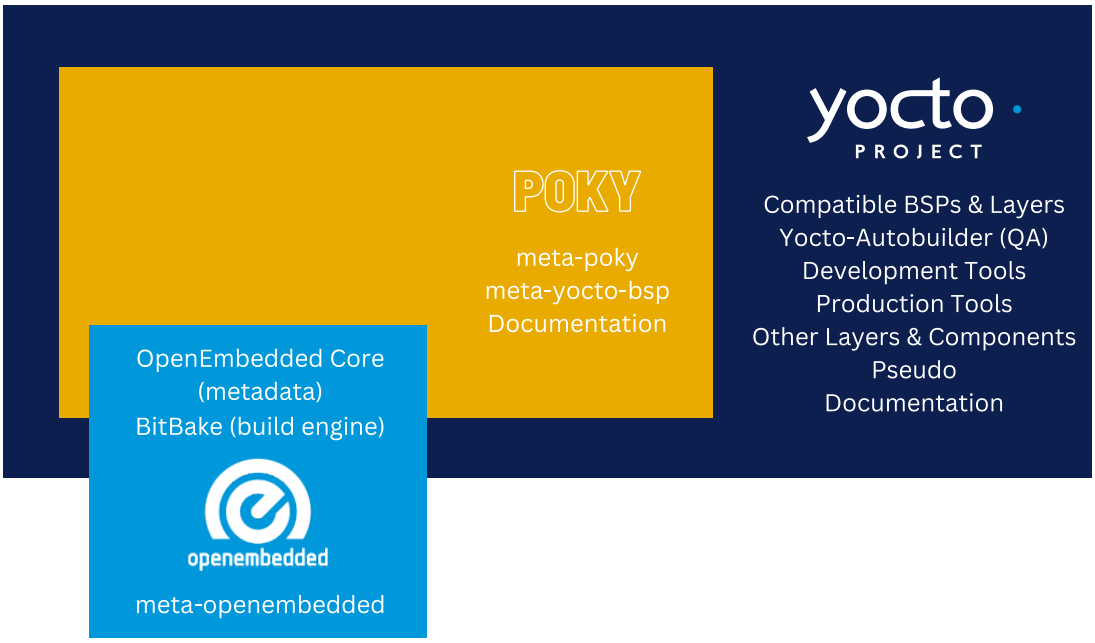
- Sistem izgradnje OpenEmbedded, su-održavan s OpenEmbedded Projektom koji se sastoji od OpenEmbedded-Core i BitBake-a.
- Referentna / primjerena konfiguracija ugrađenog Linuxa koja se koristi za testiranje (nazvana Poky).
- Proširena infrastruktura testiranja putem autobuildera na osnovi Buildbot-a.
- Integrirani alati za uspješan rad s ugrađenim Linuxom: alati za automatsku izgradnju i testiranje, procesi i standardi za definicije i razmjenu podrške za ploče, alati za analizu sigurnosti i usklađenost s licencama, podrška za softverske manifeste (SBoM) u SPDX-u.

Ovaj proces omogućuje razvoj prilagođenih operativnih sustava za ugrađene sustave putem, pružajući korisnicima fleksibilnost i kontrolu nad svakim korakom razvoja.

5.2.1. Proces izgradnje kroz Bitbake

Bitbake je alat za izgradnju koji se koristi unutar Yocto Project-a za automatizaciju procesa izgradnje softvera za ugrađene sustave. On omogućava programerima da definiraju kako bi se različiti dijelovi softvera trebali preuzimati, konfigurirati, kompajlirati i instalirati. Bitbake koristi recepte, koji su jednostavne tekstualne datoteke, za opisivanje svakog koraka potrebnog za izgradnju softvera. Ovaj alat podržava paralelne gradnje, upravljanje ovisnostima i ponovnu upotrebu koda, što ga čini ključnim dijelom ekosustava Yocto Project-a [22].

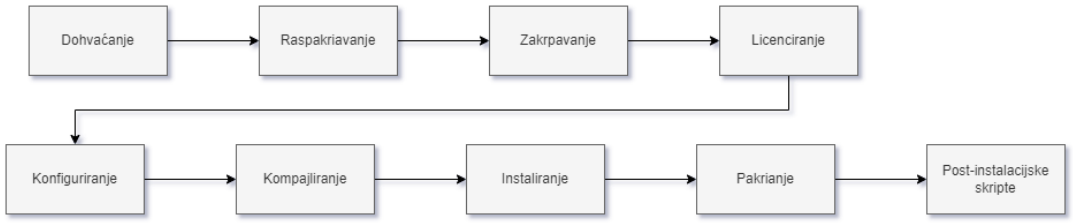
Bitbake proces izgradnje djeluje u devet koraka, cijeli proces vidljiv je i na slici 14:



Slika 13: Pregled Yocto Projekta [22]

1. dohvati (eng. *fetch*)
2. raspakiraj (eng. *unpack*)
3. zakrpi (eng. *patch*)
4. licenciraj (eng. *licensing*)
5. konfiguriraj (eng. *configure*)
6. kompajliraj (eng. *compile*)
7. instaliraj (eng. *install*)
8. pakiraj (eng. *package*)
9. post-instalacijski skripte (eng. *post-install scripts*)

Nakon što je recept konstruiran, prvi korak je preuzimanje datoteka projekta. Bitbake podržava većinu uobičajenih protokola, uključujući git, HTTPS, FTP i file za lokalne datoteke. Fetcher se pokreće uključivanjem varijable SRC URI u datoteku recepta i dodavanjem protokola prije lokacije. Kao sekundarni korak, Fetcher također izvršava provjeru kontrolne sume datoteka kako bi provjerio jesu li datoteke netaknute. Svaki preuzeti projekt ima vlastitu kontrolnu sumu koja se uključuje u datoteku recepta. U drugom koraku, razlučivanje će pokušati zaključiti je li projekt upakiran u kompresibilni format i izvršiti dekompresiju. Dodatno, razlučivanje će stvoriti novi direktorij u radnom prostoru gdje se datoteke premještaju nakon preuzimanja. Obično korisnik mora osigurati takav direktorij uključivanjem $S varijable. Međutim, za izvorne kodove u obliku tarball-a, ovo se ne zahtijeva. U trećem koraku, funkcija patch primjenjuje različite datoteke s razlikama s ekstenzijama .diff ili .patch na izvorne kodove. Svaka datoteka s razlikom mora biti ekskluzivno uključena u recept. Korak licenciranja provjerava jesu li projekti



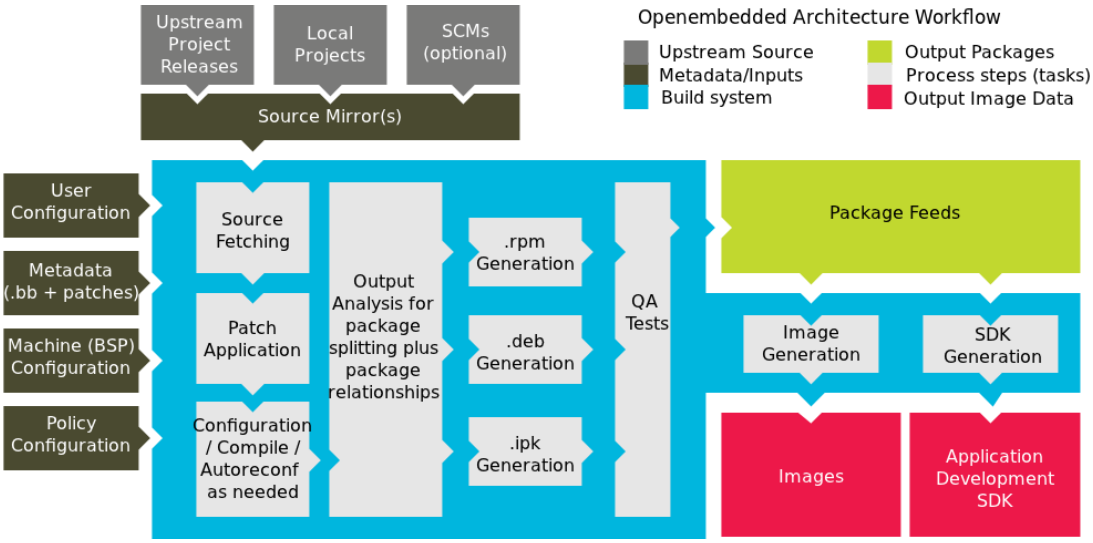
Slika 14: Bitbake proces gradnje [22]

usklađeni s bilo kojom licencom s kojom dolaze i obavezno je pružiti jednu. Kontrolna suma također mora biti navedena u datoteci licence u slučaju promjene zbog neke nadogradnje. Podržane su većina uobičajenih licenci i korisnici mogu specificirati zatvorenu licencu ako ništa nije navedeno. Licence se zatim prikupljaju u direktoriju u mapu implementacije. Tijekom konfiguracijske faze, rješavaju se sve ovisnosti o receptima. Ako su zahtjevi za ovisnosti zadovoljeni, pokušava se omogućiti bilo koje opcionalne značajke u receptu. Tijekom ove faze, razvojni programer može konfigurirati sve što je potrebno prije koraka kompilacije. Ovaj korak se može opcionalno izostaviti ako se koriste uobičajeni alati za izgradnju poput Make-a ili Cmake-a ili ako nema ništa za konfiguriranje. Kompilacijski korak koristi se za kompiliranje i povezivanje projekta. Sva potrebna konfiguracija treba biti završena prije ovog koraka i pronađene su ovisnosti. U koraku instalacije, Yocto će kopirati sve rezultirajuće datoteke iz izvora, izgradnje ili radnog direktorija i dodati ih u sliku. Ovaj dio obično se može izostaviti iz recepata prilikom izgradnje s CMake-om. Ako su potrebne dodatne datoteke za dovršetak instalacije, korisnik može dodati nove korake u proces izgradnje. Inače, potrebno je specificirati koje datoteke se kopiraju u konačnu sliku. Pakiranje ima više manjih zadataka koje će Yocto pokušati obaviti na rezultirajućim binarnim datotekama. Yocto će pokušati odvojiti debug i release binarne datoteke na njihova vlastita mjesta.Razne provjere kvalitete obavljaju se kako bi se osigurala ispravna funkcionalnost, poput uključenih ovisnosti za vrijeme izvršavanja. Nadalje, bilo koji korisnički post-instalacijski skripti mogu biti pokrenuti tijekom ovog koraka. Na posljetku nakon završetka svih koraka, stvara se slika korijenskog datotečnog sustava. Proces bitbake-a prema zadanim postavkama stvara pet datoteka: kernel sliku, sliku root datotečnog sustava, tarball kernel modula, bootloader i simboličke veze.

5.2.2. Tijek rada arhitekture OpenEmbedded-a

Bitbake je samo manji dio veće arhitekture OpenEmbedded projekta. Slika 15 prikazuje arhitekturu više apstrakcije. Plavo obojeni dio predstavlja sustav gradnje, odnosno Bitbake, i može se primijetiti kako tijek unutar plavo označene komponente odgovara opisanom procesu izgradnje u prošlom potpoglavlju. Osim toga, OpenEmbedded Build System koristi sveobuhvatni tijek rada za generiranje slika i SDK-ova, omogućujući developerima da specificiraju sve potrebne detalje za uspješnu izgradnju softvera za ugrađene sustave.

Ovo je moguće zbog robusne i kompleksne meta-strukture projekta i projektnih datoteka te brojnih dijelova koje utječu na cjelokupni proces. Ključni pojmovi uključuju metapodatke (eng. *Metadata*), koji su temelj Yocto projekta i obuhvaćaju recepte, konfiguracijske datoteke



Slika 15: Arhitektura OpenEmbedded tijeka rada [22]

i druge informacije potrebne za izgradnju slike operacijskog sustava. Recepti (eng. *Recipes*) sadrže detaljne upute za izgradnju, kompajliranje binarnih datoteka ili drugih paketa. Ovo uključuje izvore, zakrpe, ovisnosti i kompilacijske opcije. Slojevi (eng. *Layer*) grupiraju povezane recepte, omogućujući prilagodbu gradnje za različite arhitekture i bolje upravljanje metapodacima. Paketi (eng. *Packages*) su kompajlirani izlazi recepata, koji se koriste za instalaciju i izvršavanje na ciljnoj platformi. Konfiguracijske datoteke (eng. *Configuration Files*) definiraju globalne varijable i hardverske postavke, određujući što će biti uključeno u gradnju slike. OpenEmbedded-core (OE-core) je centralni skup validiranih recepata i klasa zajedničkih za mnoge OpenEmbedded sustave, osiguravajući kvalitetu i stabilnost gradnje. Na kraju, Poky je referentna distribucija koja služi kao polazna točka za prilagodbu i testiranje Yocto projekta, pružajući osnovnu funkcionalnu distribuciju i okvir za daljnji razvoj. Poky sadrži već spomenuti Bitbake.

Cijeli radni tijek može se opisati u sljedećih nekoliko koraka:

1. Razvojni inženjeri specificiraju arhitekturu, politike, zakrpe i konfiguracijske detalje.
2. Sustav gradnje (*Bitbake*) dohvaća i preuzima izvorni kod s navedene lokacije. Sustav podržava standardne metode kao što su tarballs ili sustavi za upravljanje izvornim kodom kao što je Git.
3. Nakon preuzimanja izvornog koda, sustav gradnje izvlači izvore u lokalno radno područje gdje se primjenjuju zakrpe te se pokreću zajednički koraci za konfiguriranje i kompajliranje softvera.
4. Sustav gradnje zatim instalira softver u privremeno područje za postavljanje gdje se koristi odabrani binarni paketni format (DEB, RPM ili IPK) za spremanje softvera.
5. Različite provjere kvalitete i provjere ispravnosti (QA) se provode tijekom cijelog procesa gradnje.

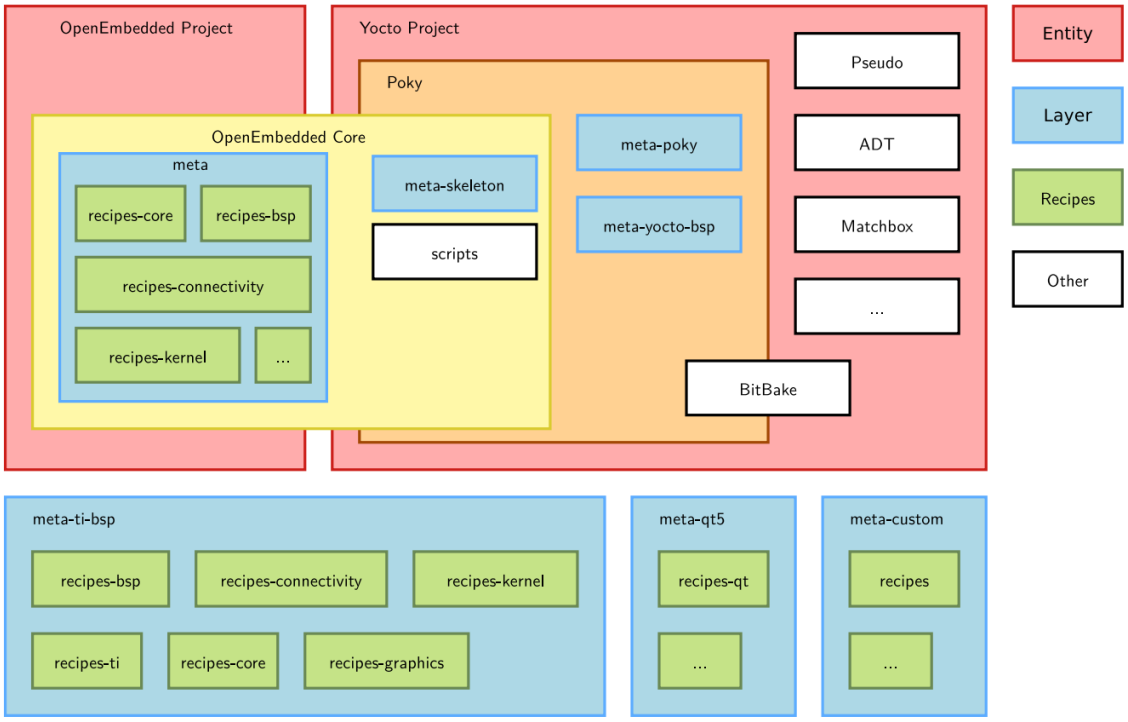
6. Nakon što se binarni paketi stvore, sustav gradnje generira binarni paketni feed koji se koristi za stvaranje konačne slike root datotečnog sustava.

7. Sustav gradnje generira sliku datotečnog sustava i prilagođeni Extensible SDK (eSDK) za razvoj aplikacija paralelno.

5.2.3. Primjer projektne strukture

Slika 16 prikazuje primjer jednog Yocto projekta, ilustrirajući kako su različiti slojevi, entiteti i recepti organizirani unutar arhitekture Yocto i OpenEmbedded (OE) projekata. U sredini slike nalaze se dva ključna entiteta: OpenEmbedded Project i Yocto Project. OpenEmbedded Project (OE) uključuje već spomenuti OE-Core, koji je centralna komponenta koja se također dijeli s Yocto Project-om. OE-Core osigurava temeljne recepte i konfiguracijske datoteke koje omogućuju osnovne funkcionalnosti i stabilnost, čime se olakšava razvoj prilagođenih Linux-baziranih sustava. Na slici je vidljivo koje komponente sadržava Poky. Poky sadrži OE-Core, zajedno sa specifičnim slojevima kao što su meta-poky i meta-yocto-bsp. Ovi slojevi omogućuju prilagodbu i proširenje osnovne funkcionalnosti, pružajući dodatne recepte i konfiguracijske opcije za razvoj specifičnih ugrađenih sustava. Na slici se također prikazuju različiti slojevi kao što je "meta", koji uključuje recepte za različite komponente (npr. recipes-core, recipes-bsp, recipes-connectivity, recipes-kernel). Ovi slojevi omogućuju modularni pristup razvoju, gdje se specifične funkcionalnosti mogu dodavati ili uklanjati prema potrebi projekta. Na primjer, sloj "meta-ti-bsp" sadrži recepte specifične za Texas Instruments hardverske platforme, dok "meta-qt5" pruža recepte za integraciju Qt grafičkog korisničkog sučelja. Bijeli elementi poput "scripts" i alata kao što su "Bitbake", "Pseudo", "ADT" i "MatchBox" podržavaju cijeli ekosustav, omogućujući automatizaciju procesa izgradnje, testiranje, upravljanje licencama i druge kritične zadatke. Bitbake je ključni alat za automatizaciju procesa izgradnje, omogućujući programerima da definiraju kako bi se različiti dijelovi softvera trebali preuzimati, konfigurirati, kompajlirati i instalirati.

Ovaj primjer arhitekture pokazuje fleksibilnost i skalabilnost Yocto Project-a, gdje razvojni inženjeri mogu koristiti i prilagoditi različite slojeve i recepte kako bi stvorili prilagođene Linux distribucije za specifične ugrađene sustave. Primjer pokazuje kako se različiti slojevi i entiteti međusobno nadopunjuju, omogućujući suradnju i dijeljenje resursa između Yocto Project-a i OpenEmbedded Project-a, čime se postiže veća efikasnost i ponovna upotreba koda u razvoju ugrađenih sustava. Kako bi se ovaj projekt proširio, potrebno je jednostavno nadopuniti meta-custom sloj s dodatnim receptima. Tako je, na primjer, moguće napisati recepte za gradnju aplikacija, konfiguracije i sl. Osim recepta moguće je naravno izmijeniti već postojeće recepte ili dodati novi sloj koji će se integrirati u proces gradnje. Prednost Yocto projekta je što je mnogo meta-slojeva za određene ploče, projekte ili programe već dostupno te ih je samo potrebno dodati na ovu arhitekturu te konfigurirati/modificirati željene postavke. Primjeri recepata i konkretne strukture detaljno su objašnjeni u sljedećem poglavlju.



Slika 16: Primjer jednog Yocto projekta [23]

5.3. Usporedba Buildroot i Yocto razvojne okoline

Usporedba Buildroota i Yocto razvojnih okvira otkriva nekoliko značajnih razlika u njihovim pristupima, posebno u strukturi projekta i procesu izgradnje operativnih sustava za ugrađene sustave.

U Buildrootu, proces izgradnje je pojednostavljen i automatiziran putem alata koji omogućuje generiranje cijelog Linux sustava za ugrađene uređaje korištenjem križne kompilacije. Ovaj alat olakšava konfiguraciju i izgradnju osnovnih dijelova arhitekture Linuxa, kao što su korijenski datotečni sustav, slika jezgre Linuxa te alatni lanac za križnu kompilaciju. Buildroot se može prilagoditi raznim kombinacijama ovih opcija i podržava različite procesore poput ARM-a i MIPS-a, te dolazi s osnovnim konfiguracijama za nekoliko standardnih ploča dostupnih na tržištu. Iako nudi jednostavan proces konfiguracije, Buildroot se suočava s izazovima u razdvajanju novih projekata od zadanih. Drugim riječima potrebna je cijela prekonfiguracija ako se s određenim projektom želi postići nešto drugo. Za razliku od tog aspekta buildroot-a, Yocto Project pruža fleksibilnost i robustnost u konfiguriranju slika operativnih sustava za ugrađene uređaje. Iako nudi veću robusnost u upravljanju ovisnostima između projekata i definiranju strukture direktorija, Yocto dolazi s većom složenošću i strmijom krivuljom učenja. Koristi više datoteka, lokacija datoteka i sintaksa za opisivanje konfiguracije projekta, što može otežati koordinaciju u većim poduzećima. Unatoč izazovima s kojima se suočava, Yocto nudi mogućnosti upravljanja projektima i pouzdaniju kompilaciju, posebno u upravljanju ovisnostima tijekom izvršavanja.

Što se tiče vremena izgradnje, oba sustava imaju svoje prednosti i izazove. Buildroot omogućuje potencijalno kraće iteracije zahvaljujući manjom početnom konfiguracijom

[24]. Yocto s druge strane proizvede veće datoteke na projektnom računalu te kao takav možda nije prikladan za uređaje s ograničenom memorijom. Međutim, Yocto nudi više mogućnosti za upravljanje projektima i kompleksnijim scenarijima.

Ukratko, Buildroot nudi jednostavniji pristup izgradnji operativnih sustava za ugrađene uređaje, dok Yocto pruža veću fleksibilnost i kontrolu, ali uz veću složenost i krivulju učenja. Ove prednosti i mane vidljive su i u tablici 6. Odabir između ova dva okvira ovisi o specifičnim potrebama i zahtjevima projekta. Ako je potrebno brzo napraviti jednostavu sliku operacijskog sustava, buildroot će jednostavnije postići taj zadatak. Ukoliko je plan držati produkcijski lanac s održavanjem više tehnologija, Yocto je prikladniji.

Tablica 6: Usporedba razvojnih okolina Buildroot i Yocto[24]

	Yocto	**Buildroot**
Prednosti	• Robusna konfiguracija ovisnosti • Dobro definirana struktura projekta • Mogućnost upravljanja ovisnostima tijekom izvršavanja	• Jednostavnost korištenja • Potencijalno brže vrijeme izgradnje • Pogodan za ugradbene uređaje s ograničenom memorijom
Mane	• Složenost i strmija krivulja učenja • Potreba za više resursa za početno konfiguriranje • Veći vremenski zahtjevi za izgradnju	• Manje robusna konfiguracija ovisnosti • Izazovi u razdvajanju novih projekata od zadanih • Manje mogućnosti upravljanja projektima

6. Praktična implementacija Yocto projekta za Raspberry Pi s grafičkim sučeljem

Za potrebe ovog rada, kreirana je prilagođena linux slika s pomoću Yocto projekta s minimalnom konfiguracijom koja uključuje *QT* i *Weston* za grafičku aplikaciju. Ova slika je instalirana na Raspberry Pi 4b (Raspi). Na GPIO pinove Raspberry Pi-ja spojen je mali DHT22 senzor. Aplikacija omogućava pregled trenutnog stanja senzora te prikaz statistika temeljenih na prikupljenim podacima pohranjenim u *JSON* formatu. Slika 17 prikazuje postavu projekta. Raspi mora biti spojen na nekakav ekran kako bi prikazivao radnju aplikacije. Iako mnogim IoT uređajima nije potrebno grafičko sučelje, ovaj projekt je dobra demonstracija što je potrebno kako bi se grafičko sučelje ostvarilo te daje predodžbu o cijelom procesu te konačnoj veličini slike operacijskog sustava. Cijeli projektni direktorij s implementacijama i drugim povezanim repozitorijima dostupan je na githubu [25].

Praktični dio rada obuhvaća nekoliko ključnih koraka:

- **Postavljanje razvojne okoline**: Konfiguracija potrebnih alata kao što su Git, Yocto, Qt Creator i potrebnih biblioteka za križnu kompilaciju.
- **Razvoj i konfiguracija Yocto projekta**: Kreiranje prilagođene Linux distribucije s Yocto Projectom, uključujući pisanje recepata i konfiguracijskih datoteka za specifične potrebe projekta.
- **Razvoj grafičke aplikacije**: Korištenje Qt Creatora za razvoj aplikacije koja vizualizira podatke dobivene od DHT22 senzora. Aplikacija je dizajnirana da bude intuitivna i korisniku pruža jasne informacije o temperaturi i vlažnosti.
- **Testiranje i optimizacija**: Iterativno testiranje razvijenih rješenja na Raspberry Pi platformi, otklanjanje grešaka i optimizacija performansi sustava.

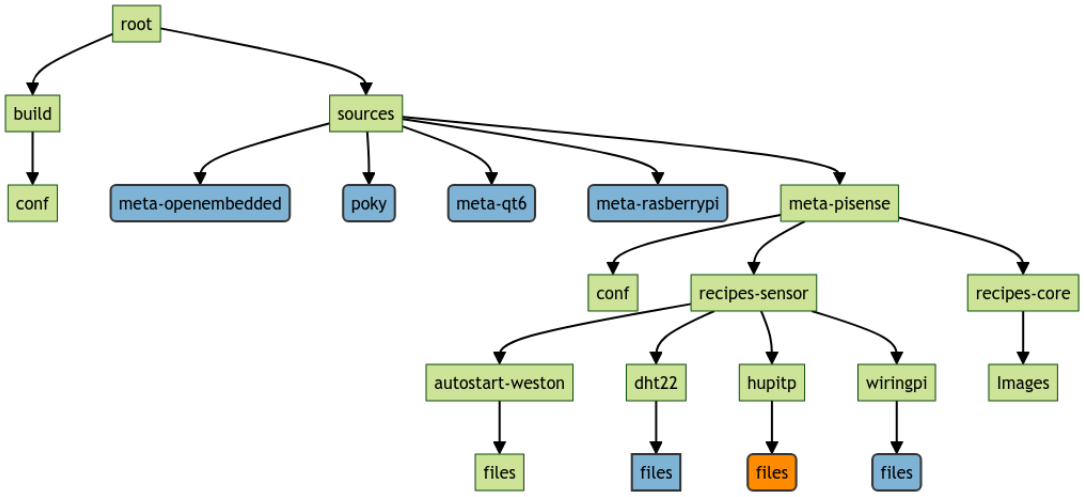
6.1. Projektna struktura

Cijela projektna struktura prikazana je na slici 17. Build direktorij sadržava sve lokalne datoteke potrebne za gradnju slike operacijskog sustava. Ona je srž procesa izgradnje u Yocto projektu, gdje se definiraju i konfiguriraju različite komponente kako bi se generirala prilagođena Linux distribucija. Iako cijeli taj direktorij nije uključen u git, najvažniji dio, *conf/local.conf*, je uključen. Ova konfiguracijska datoteka predstavlja srce projekta, gdje se definiraju varijable, postavke okoline i ostali parametri koji utječu na izgradnju slike operacijskog sustava. Kada se aktivira okruženje za *poky*, lokalna konfiguracija se obično generira automatski ako već ne postoji. Ovaj proces uključuje stvaranje ili kopiranje osnovne konfiguracijske datoteke *conf/local.conf* u korijenskom direktoriju projekta. Ta osnovna konfiguracijska datoteka sadrži minimalne postavke potrebne za pokretanje procesa izgradnje, poput varijabli za odabir distribucije, arhitekture, verzije alata za izgradnju i slično. Osim *local.conf* važna datoteka konfiguracije je i *bblayers.conf* koja definira koji slojevi su uključeni te prioritet slojeva.

U direktoriju *"sources"* vidljivi su nekoliko meta slojeva. Plavo označeni pravokutnici označavaju git podmodule (*eng. git submodule*), što su linkovi reference na druge git repozitorije koji sadrže različite komponente potrebne za izgradnju Linux distribucije. Na razini meta slojeva su integrirani esencijalni slojevi *meta-openembedded* i *poky*. Osim njih, potrebno je bilo uključiti meta-rasberrypi sloj koji sadržava konfiguracije, pogonske programe i postavke specifične za RaspberryPi uređaje. Kako bi se mogla pokrenuti grafička aplikacija, bitno je uključiti i sloj *meta-qt6*. Ovaj sloj sadržava biblioteke za pokretanje i kompiliranje Qt aplikacija. Qt je popularan i snažan framework za gradnju grafičkih aplikacija u jeziku C++, u kojem je aplikacija pisana.

Meta-pisense je autorski definirani sloj za potrebe ove praktične implementacije. Ovaj sloj sadrži *local/conf* za lokalnu konfiguraciju sloja te različite autorski definirane recepte. Tako *recipes-core* sadržava recept za gradnju prilagođene slike operacijskog sustava koja je na kraju instalirana na Raspi uređaj, dok *Recipes-sensor* sadržava sve potrebne biblioteke i druge skripte potrebne za izvođenje glavnog dijela slike, odnosno autorske konfiguracije i aplikacije *hupitp*. Ovo uključuje grafičku aplikaciju, potrebnu biblioteku *wiringpi* te recept *autostart-weston* koja sadrži skripte za automatski pokretanje grafičkog sučelja i pokretanje aplikacije.

Narančasto označeni pravokutnik pod *hupitp* ukazuje na autorski git repozitorij. Ovaj repozitorij sadržava implementaciju grafičke aplikacije koja je ključna za funkcionalnost projekta. Kako bi aplikacija radila, bitna je biblioteka *wiringPi*, koja omogućava pristup GPIO pinovima na Raspberry Pi-ju. *Dht22* se ne gradi u slici, ali je uključena u projekt radi testiranja. *Dht22* također zahtijeva *wiringPi* biblioteku, a uključivanje senzora u projekt omogućava provjeru funkcionalnosti prije detaljnijeg razvoja aplikacije *HuPiTp*-a. Ova struktura osigurava temelje za razvoj, testiranje i konačnu implementaciju kompleksnih funkcionalnosti u Yocto projektu za Raspberry Pi s grafičkim sučeljem.



Slika 17: Struktura direktorija projeknte implementacije; Vlastiti rad

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

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

BBLAYERS ?= " \
    ${TOPDIR}/../sources/meta-openembedded/meta-oe \
    ${TOPDIR}/../sources/meta-openembedded/meta-python \
    ${TOPDIR}/../sources/poky/meta \
    ${TOPDIR}/../sources/poky/meta-poky \
    ${TOPDIR}/../sources/meta-qt6 \
    ${TOPDIR}/../sources/meta-pisense \
    ${TOPDIR}/../sources/meta-raspberrypi \
"
```

Isječak koda 1: Sadržaj datoteke `bblayers.conf`

6.2. Konfiguracijske datoteke

U konfiguracijskim datotekama bblayers.conf i local.conf definiraju se različite postavke za izgradnju Yocto projekta. Te datoteke igraju ključnu ulogu u procesu izgradnje prilagođene Linux distribucije za Raspberry Pi uređaje s grafičkim sučeljem.

U prošlom poglavlju već je spomenuta važnost datoteke bblayers.conf za projekt Yocto. Ona određuje koji su svi slojevi uključeni u projekt te njihov redoslijed. Uključivanje određenih slojeva, poput meta-raspberrypi i meta-qt6, ključno je za podršku specifičnim hardverskim i softverskim zahtjevima Raspberry Pi platforme te za integraciju Qt grafičkog okvira. Ako se neki sloj izostavi iz ove datoteke, taj sloj neće biti uključen u sliku operacijskog sustava. Važno je napomenuti da se varijabla *POKY_BBLAYERS_CONF_VERSION* povećava svaki put kada se datoteka *bblayers.conf* promijeni na način koji nije kompatibilan s prethodnim verzijama. Ovo je korisno za praćenje promjena u datoteci i osiguravanje da se koriste ispravne verzije. Dodatno, pri odabiru slojeva za projekt, preporučuje se korištenje onih koji su već testirani i podržani od strane zajednice. To može smanjiti moguće probleme i olakšati integraciju novih komponenti. Tako se korisnik može fokusirati samo na pisanje vlastitog sloja s receptima za dijelove projekta koji su njemu važni.

Dodatno, za uključivanje ili kreiranje sloja koristi se naredba bitbake-layers. Na primjer, za dodavanje novog sloja u projekt može se koristiti sljedeća bitbake naredba (nakon što je aktivirano bitbake okruženje):

```
bitbake-layers add-layer /putanja/do/novog/sloja
```

Datoteka ‘local.conf‘ je lokalna konfiguracijska datoteka u kojoj se mogu postaviti različite opcije specifične za projekt. Sadržaj datoteke glavne projektne konfiguracije *local.conf* prikazana je u programskom isječku 2:

U ovoj datoteci postavljaju se različite opcije poput značajki slike, postavki okruženja, varijabli okoline i slično. Postavka `IMAGE_FEATURES += "allow-root-login"` omogućuje `root` korisniku prijavu na sustav. To može biti korisno za razvojne svrhe, iako se

```
IMAGE_FEATURES += " allow-root-login"
QT_QPA_PLATFORM = "wayland"
XDG_RUNTIME_DIR = "/run/user/${UID}"
LANG="C.UTF-8"
QT_QPA_FONTDIR="/usr/share/fonts/ttf"
MACHINE ??= "raspberrypi4-64"
LICENSE_FLAGS_ACCEPTED = "synaptics-killswitch"
ENABLE_UART= "1"
#SSTATE_DIR ?= "${TOPDIR}/sstate-cache"
#TMPDIR = "${TOPDIR}/tmp"
#DL_DIR ?= "${TOPDIR}/downloads"
DISTRO ?= "poky"
DISTRO_FEATURES:append = " wayland opengl"
```

Isječak koda 2: Sadržaj datoteke `local.conf`

preporučuje ograničiti `root` pristup u produkcijskim okruženjima zbog sigurnosnih razloga. `QT_QPA_PLATFORM = "wayland"` definira da će Qt koristiti Wayland kao platformu za prikaz. Wayland je suvremeni zamjenik za X11 sustav koji omogućuje bolju podršku za moderne grafičke funkcionalnosti.

Varijabla `XDG_RUNTIME_DIR = "/run/user/$UID"` definira direktorij za pohranu runtime datoteka specifičnih za korisnika. `$UID` predstavlja korisnički ID koji se dinamički zamjenjuje prilikom izvršavanja. Postavka `LANG = "C.UTF-8"` postavlja zadani jezični i lokacijski format na UTF-8, što osigurava pravilno rukovanje tekstom i simbolima u različitim jezicima. `QT_QPA_FONTDIR = "/usr/share/fonts/ttf"` definira putanju do direktorija gdje se nalaze TrueType fontovi koje koristi Qt za prikaz teksta. Varijabla `MACHINE ??= "raspberrypi4-64"` definira ciljnu platformu za izgradnju, u ovom slučaju 64-bitnu verziju Raspberry Pi 4.

`LICENSE_FLAGS_ACCEPTED = "synaptics-killswitch"` postavka prihvaća specifične licence koje su potrebne za izgradnju određenih komponenti, u ovom slučaju za komponente koje zahtijevaju `synaptics-killswitch` licencu. `ENABLE_UART = "1"` omogućuje UART (*eng. Universal Asynchronous Receiver/Transmitter*) komunikaciju, što je korisno za serijsku komunikaciju s Raspberry Pi uređajem. Ovo služi za jednostavnije debuganje i pristup uređaju. Dodatne korisne postavke uključuju `SSTATE_DIR ?= "$TOPDIR/sstate-cache"`, koja određuje direktorij za sstate cache. Sstate cache je mehanizam koji omogućuje bržu ponovnu izgradnju tako da sprema rezultate prethodnih gradnji. `TMPDIR = "$TOPDIR/tmp"` definira direktorij za privremene datoteke koje nastaju tijekom procesa izgradnje. Zadana vrijednost je direktorij `tmp` unutar `TOPDIR`. Varijabla `DL_DIR ?= "$TOPDIR/downloads"` određuje direktorij gdje će BitBake pohraniti preuzete datoteke, poput izvornog koda i binarnih datoteka. Sva tri ova direktorija su postavljena na zadane vrijednosti, ali su radi pojašnjena uključena u ovaj dio teksta, zato u 2 imaju # ispred.

Varijabla `DISTRO ??= "poky"` definira distribuciju koja se koristi kao osnovna konfiguracija za izgradnju. Poky je referentna distribucija u Yocto projektu. Jedna od značajnijih postavka je `DISTRO_FEATURES = " wayland opengl"` koja dodaje dodatne značajke distribuciji, u ovom slučaju omogućujući podršku za Wayland i opengl. Wayland je moderni protokol za prikaz koji zamjenjuje X11. Wayland omogućuje naprednije grafičke mogućnosti,

bolju izvedbu i sigurnost te je posebno koristan za uređaje s ograničenim resursima, kao što je Raspberry Pi. OpenGL (*eng. Open Graphics Library*) omogućuje hardversko ubrzanje grafike. OpenGL je standardna specifikacija za prikazivanje 2D i 3D grafike. Omogućavanjem ove značajke, distribucija može koristiti hardverske mogućnosti Raspberry Pi uređaja za renderiranje grafike, što rezultira boljom izvedbom grafičkih aplikacija i igara. OpenGL podrška je ključna za aplikacije koje zahtijevaju visokokvalitetnu grafiku i visoku izvedbu.

6.3. Aplikacijski sloj

Sloj `meta-pisense` predstavlja autorski definirani sloj unutar Yocto projekta razvijenog za potrebe ovog rada. Ovaj sloj obuhvaća sve specifične konfiguracije i recepte potrebne za prilagođenu Linux distribuciju s grafičkim sučeljem za Raspberry Pi 4b. Kroz `meta-pisense`, integrirane su ključne komponente poput prilagođenih recepata za senzorske aplikacije, potrebnih biblioteka te skripti za automatsko pokretanje grafičkog sučelja. Sloj je strukturiran kako bi omogućio lako prilagođavanje i proširenje funkcionalnosti, osiguravajući da svi potrebni elementi za rad aplikacije, uključujući QT framework i podršku za senzore, budu pravilno konfigurirani i integrirani.

6.3.1. Recept slike operacijskog sustava

Recept slike operacijskog sustava dostupan je pod putanjom u *source* direktoriju *meta-pisense/recipes-core/images/pisense.bb*. Slika je nazvana *pisense* kao Pi senzor. Ovaj recept definira sve potrebne komponente i postavke za izradu prilagođene Linux distribucije koja se koristi za prikupljanje podataka o temperaturi i vlažnosti zraka pomoću senzora te prikazivanje tih podataka putem jednostavnog grafičkog sučelja. Isječak koda ovog recepta vidljiv je u isječku 3.

Recept započinje opisom slike (`DESCRIPTION`), nakon čega je definirana licenca. Ova slika koristi licencu `MIT`, što je definirano kroz varijablu `LICENSE`, dok je `LIC_FILES_CHKSUM` varijabla korištena za provjeru ispravnosti licencne datoteke.

Nasljeđivanjem `core-image` klase (`inherit core-image`), ovaj recept preuzima osnovne funkcionalnosti potrebne za izradu osnovne slike operacijskog sustava. Uključene su različite značajke slike putem varijable `IMAGE_FEATURES`, koja uključuje `splash` za prikazivanje splash screena prilikom pokretanja, `package-management` za upravljanje paketima, `ssh-server-dropbear` za omogućavanje SSH pristupa, `hwcodecs` za hardversko dekodiranje videozapisa i `weston` za pokretanje Weston kompozitora, koji je neophodan za Wayland grafičko sučelje.

Kako bi se omogućila dodatna sigurnost i prilagodba korisnika, recept nasljeđuje i `extrausers` klasu (`inherit extrausers`). Ova klasa omogućuje postavljanje početnih korisničkih podataka, uključujući zaporku za `root` korisnika definiranu kroz varijablu `PASSWD`. `EXTRA_USERS_PARAMS` varijabla koristi se za promjenu `root` zaporke, što je posebno korisno za razvojne svrhe.

```
DESCRIPTION = "Custom Raspberry Pi image with simple GUI for collecting temperature
↪  and humidity data"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
↪  "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

inherit core-image

IMAGE_FEATURES += "splash package-management ssh-server-dropbear hwcodecs weston "

inherit extrausers
PASSWD = "passowrdHash"
EXTRA_USERS_PARAMS = "\
    usermod -p '${PASSWD}' root; \
    "

# Add your custom application package to be installed
IMAGE_INSTALL += "raspi-gpio \
          dht22 \
          hupitp \
          ${Qt_UTILS}"

Qt_UTILS = "qtbase \
            qtquick3d \
            qtdeclarative \
            qtwayland \
            qtgraphs"

RDEPENDS_${PN} = "qtwayland"
PACKAGECONFIG += " qtwayland wayland libinput fontconfig"

QT_QPA_PLATFORM = "wayland"

QB_MEM = "-m 512"
```

Isječak koda 3: Recept za izgradnju slike operacijskog sustava `pisense`

Varijabla IMAGE_INSTALL određuje koje će se dodatne aplikacije i paketi instalirati na sliku. Ovdje su uključeni raspi-gpio, alat za upravljanje GPIO pinovima na Raspberry Pi-ju (ovo je dio *meta-rasberrypi* sloja), dht22, paket koji omogućuje testiranje i rad s DHT22 senzorom (ova komponenta je opcionalna). Na kraju je uključena i hupitp, glavna aplikacija za prikupljanje i prikaz podataka sa senzora sa grafičkim sučeljem. Također, varijabla Qt_UTILS specificira Qt biblioteke koje su potrebne za grafičku aplikaciju, uključujući qtbase (osnovne Qt biblioteke), qtquick3d (za 3D grafiku), qtdeclarative (za QML), qtwayland (za Wayland podršku) i qtgraphs (za grafove).

Recept također postavlja ovisnosti (RDEPENDS_$PN) za paket, osiguravajući da su sve potrebne biblioteke i moduli instalirani. U ovom slučaju, dodana je podrška za qtwayland. PACKAGECONFIG varijabla dodatno konfigurira pakete koji će biti uključeni, u ovom slučaju tu spadaju qtwayland, wayland, libinput (za unos s različitih uređaja) i fontconfig (za upravljanje fontovima).

Varijabla QT_QPA_PLATFORM je varijabla okružja postavljena na wayland definira da će Qt koristiti Wayland kao platformu za prikaz. Konačno, QB_MEM = "-m 512" postavka osigurava da će se virtualni stroj koji se koristi za gradnju slike pokretati s 512 MB memorije, što je dovoljno za osnovne potrebe gradnje i testiranja.

Ovaj recept pruža sve potrebne konfiguracije za izradu funkcionalne slike operacijskog sustava koja podržava grafičko sučelje za prikupljanje i prikaz podataka sa senzora na Raspberry Pi-ju. Kako bi se izradila ova slika nakon što je u okruženje postavljen bitbake moguće je pozvati sljedeću komandu:

```
bitbake pisense
```

6.3.2. Recept biblioteke - WiringPi

Recept za WiringPi biblioteku, smješten u direktoriju pod sources *meta-pisense/recipes-core/wiringpi/wiringpi.bb*, omogućuje upravljanje GPIO kanalima na Raspberry Pi-ju. WiringPi je popularna biblioteka koja pruža jednostavne alate za rad s GPIO pinovima putem C/C++ jezika [26]. Ovaj specifični recept temelji se na modificiranom račvanju (*eng. forku*) originalnog WiringPi repozitorija, koji koristi CMake umjesto uobičajenih Makefile datoteka za proces izgradnje. Ovo račvanje je razvijeno u svrhu ovog rada, a autori originalnog WiringPi repozitorija su zatražili da se ova implementacija integrira u repozitorij kao zasebna CMake grana [27]. Ovaj Cijeli kod recepta vidljiv je u isječku 4.

HOMEPAGE varijabla upućuje na službenu stranicu projekta, dok SECTION varijabla specificira da biblioteka pripada razvojnim alatima i knjižnicama. LICENSE varijabla definira da je biblioteka licencirana pod MIT licencom, a LIC_FILES_CHKSUM varijabla pruža kontrolni zbroj za licencnu datoteku.

Ovaj recept koristi CMake za proces izgradnje umjesto tradicionalnih Makefile-ova, što omogućuje lakše upravljanje i prilagodbu biblioteke za bitbake. Ovo je zato što bitbake iako može koristiti bash komande, neke stvari su ograničene zbog korisnika koji ih koristi, tako

```
DESCRIPTION = "A library to control Raspberry Pi GPIO channels"
HOMEPAGE = "https://projects.drogon.net/raspberry-pi/wiringpi/"
SECTION = "devel/libs"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://COPYING.LESSER;md5=e6a600fd5e1d9cbde2d983680233ad02"

SRC_URI[md5sum] = "492aa9e6edcb42abb1e724a3bb9d11ef"
SRC_URI[sha256sum] =
↪  "a383cfa1a7bd6e1c0e96645d04d90cbeda2c62712183921b6661f02225961037"

SRC_URI = "
file://CMakeLists.txt
file://version.h
file://COPYING.LESSER
file://debian
file://debian-template
file://devLib
file://gpio
file://wiringPi/
file://wiringPiD/
"

SRC_URI[md5sum] = "c227be5416f474bd3022eb3d3fb0ee2d"
SRC_URI[sha256sum] =
↪  "32a9c2f1f992fd70b1f61de9f58733a226a68d65e947f1fe672a94b03d9c292f"

S = "${WORKDIR}/wiringPi"

inherit cmake

DEPENDS += "virtual/kernel libxcrypt"

EXTRA_OECMAKE += "-DWIRINGPI_SUDO
=OFF"

FILES_${PN} = "${libdir}/* ${includedir}/"
FILES_${PN}-dev += "${libdir}/ ${includedir}/"
FILES_${PN}-dbg += "${libdir}/"

BBCLASSEXTEND = "native"
```

Isječak koda 4: Recept za izgradnju WiringPi biblioteke

da sudo, brisanje i neke druge opcije jednostavno ne prolaze u originalnom *makefile* zapisu. Ključne datoteke i direktoriji uključeni u izgradnju su definirani u `SRC_URI` varijabli. Nakon izgradnje, datoteke biblioteke i pripadajuće header datoteke pohranjuju se u odgovarajuće direktorije definirane u `FILES_PN` varijablama. Dodatne postavke i ovisnosti specificirane su unutar recepta kako bi se osigurala ispravna izgradnja i integracija WiringPi biblioteke u ciljnu sliku operacijskog sustava. Dvije neophodne funkcije koja ova biblioteka omogućuje su *wiringPiSetupGpio()* i *digitalRead()*.

Funkcija *wiringPiSetupGpio()* se koristi za inicijalizaciju biblioteke i postavljanje rada GPIO pinova na GPIO način rada, omogućujući pristup GPIO pinovima koristeći BCM (*eng. Boradcom)* brojeve pinova. Ova funkcija je neophodna za pravilno funkcioniranje WiringPi biblioteke i pristup GPIO pinovima na Raspberry Pi-ju. Osim toga, funkcija *digitalRead()* se koristi za čitanje digitalnog ulaza s određenog GPIO pina. Ova funkcija prima broj GPIO pina kao argument i vraća trenutno stanje tog pina - 0 ako je napon na pinu nizak (LOW) i 1 ako je napon visok (HIGH). Ova funkcija je ključna za dohvaćanje podataka s DHT22 senzora ili drugih digitalnih senzora koji koriste digitalne ulaze za komunikaciju s Raspberry Pi-jem.

6.3.3. Opis recepta i ključnih funkcija grafičke aplikacije - HuPiTp

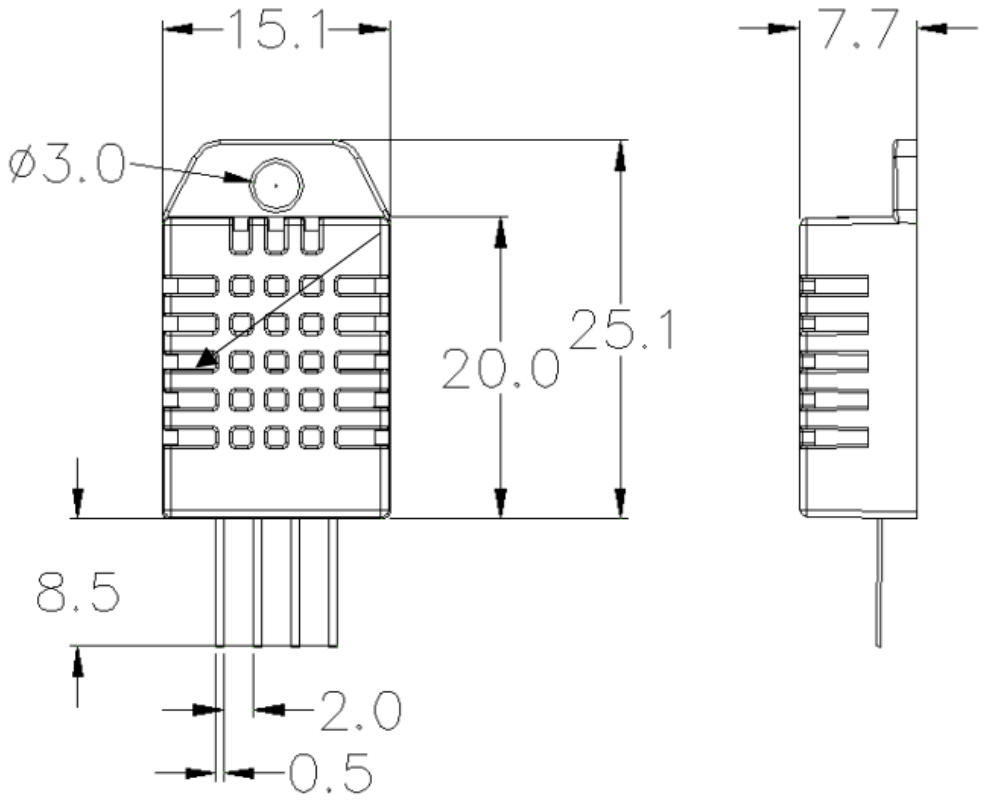
Glavna aplikacija koja daje funkcionalnost Rasberry Pi uređaju je HuPiTp skraćeno za *Humidity Pi Temperature*. Ova aplikacija dohvaća i obrađuje podatke koje prima DHT22 senzor. Ovaj senzor može detektirati trenutno stanje vlažnosti i temperature svakih 2 sekundi. Kako bi se objasnili najvažniji dijelovi programskog kôda, potrebno je prikazati kako se senzor spaja na GPIO pinove Rasberry Pi uređaja. Slika 18 prikazuje shemu DHT22 senzora dok tablica 7 opisuje koji pin služi za što (s lijeva na desno).

Tablica 7: Prikaz pinova DHT22 senzora [28]

Pin	Funkcija
1	VDD (napajanje)
2	DATA (signal)
3	NULL (ne spaja se)
4	GND (uzemljenje)

Nadalje slika 19 prikazuje shemu rasporeda pinova. Moguće je povezati ovaj uređaj na mnogo različitih načina dokle god se određeni pinovi povezuju na za to predviđena mjesta. Tako signal podataka (PIN 2) DHT22 senzora može biti spojen na bilo koji od GPIO pinova. Važno je napomenuti da GPIO pin koji se očitava, definira se u programskom kodu, stoga je važno razvojnom inženjeru da to ima na umu. Za konfiguraciju ove projektne implementacije pinovi su povezani na Rasberry Pi uređaj s pomoću DuPont kablova (*eng. jump wires*) na sljedeći način:

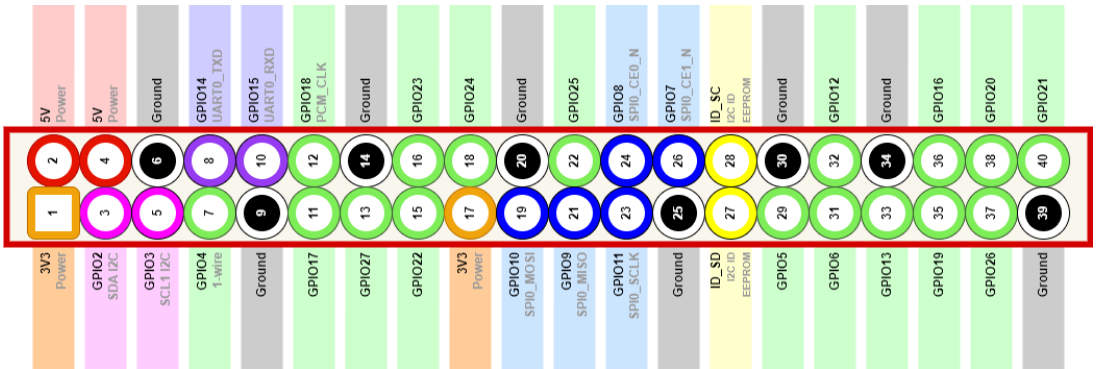
- Pin 1 DHT22 senzora na fizički Pin 1 (3v3) na Raspberry Pi uređaju.
- Pin 2 DHT22 senzora na fizički Pin 12 (GPIO18) na Raspberry Pi uređaju.



Slika 18: Shematski prikaz dht22 senzora [28]

- Pin 4 DHT22 senzora na fizički Pin 6 (GND) na Raspberry Pi uređaju.

Ova konfiguracija osigurava pravilno napajanje i komunikaciju senzora s Raspberry Pi-jem, omogućujući aplikaciji HuPiTp da prikuplja podatke o vlažnosti i temperaturi.



Slika 19: Shematski prikaz rasporeda pinova uređaja Rasberry Pi [29]

6.3.3.1. Upravitelj podataka senzora DHT22 - Klasa

Klasa `DHT22` je ključna komponenta aplikacije HuPiTp koja omogućuje dohvaćanje podataka s DHT22 senzora. Definirana je u datoteci `dht22.h` s implementacijom u `dht22.cpp`.

```
#ifndef DHT22_H
#define DHT22_H

#include
#include  // Include the necessary header
#include
#include
#include
#include
#include

class DHT22 : public QObject
{
Q_OBJECT
public:
explicit DHT22(QReadWriteLock *fileMutex, QObject *parent = nullptr);

signals:
void temperatureUpdated(float celsius, float fahrenheit);
void humidityUpdated(float humidity);

public slots:
void readAndOutputSensorDataAsJson(const QString &filename);

private:
static const unsigned short signal = 18; // GPIO pin na kojem je spojen DHT22 senzor
unsigned short data[5] = {0, 0, 0, 0, 0};

short readData();
QReadWriteLock *lock;
};

#endif // DHT22_H
```

Isječak koda 5: Definicija klase `DHT22`

Ova klasa je odgovorna za čitanje podataka sa senzora, obradu tih podataka i njihovo prosljeđivanje aplikaciji putem signala. Isječak kôda 5 prikazuje `dht22.h`, a cijeli repozitorij s kompletnom implementacijom dostupan je na githubu [30]. Glavne funkcije ove klase su:

- `wiringPiSetupGpio()`: Inicijalizira WiringPi biblioteku za korištenje s GPIO pinovima Raspberry Pi uređaja.
- `digitalRead()`: Očituje stanje na određenom GPIO pinu, što je ključno za dohvaćanje podataka s DHT22 senzora.
- `readAndOutputSensorDataAsJson()`: Čita podatke sa senzora i formatira ih u JSON oblik za daljnju obradu ili pohranu.

Kôd za klasu `DHT22` je sljedeći:

Ova klasa koristi Qt okvir za signal-slot mehanizam, što omogućuje jednostavno ažuriranje podataka i njihovo i komunikaciju s drugim dijelovima aplikacije. Definirani signali su `temperatureUpdated` i `humidityUpdated`. Funkcije `wiringPiSetupGpio()` i `digitalRead()` su ključne za inicijalizaciju i očitavanje podataka s GPIO pinova, dok `readAndOutputSensorDataAsJson()`

```
// Create JSON object
QJsonObject json;
QString dateKey = dateTime.toString("yyyy-MM-dd"); // Use the date as the key
QJsonObject entry;
entry["time"] = dateTime.toString("HH:mm
");
entry["humidity"] = humidity;
entry["temp_C"] = celsius;
entry["temp_F"] = fahrenheit;

// Read existing data from the file

// Add the new entry to the existing data
QJsonArray dateEntries = existingData.value(dateKey).toArray();
dateEntries.append(entry);
existingData[dateKey] = dateEntries;
{
QWriteLocker locker(lock);
// Write updated data back to the file
if (file.open(QIODevice::WriteOnly)) {
QJsonDocument doc(existingData);
file.write(doc.toJson());
file.write("\n"); // Add newline for readability
file.close();
} else {
fprintf(stderr, "Error opening file for writing.\n");
}
}

emit temperatureUpdated(celsius, fahrenheit);
emit humidityUpdated(humidity);
```

Isječak koda 6: Obrada podataka DHT22 senzora

pruža način za formatiranje i pohranu tih podataka u JSON formatu, što je vrlo korisno za kasniju analizu ili prikaz podataka. Varijabla `signal` je statička konstanta koja predstavlja GPIO pin na kojem je spojen DHT22 senzor. U ovom slučaju, definirana je kao `signal = 18`, što odgovara GPIO 18 odnosno fizičkom pinu 12 na Raspberry Pi uređaju.

Isječak kôda 6 prikazuje dio funkcije *readAndOutputSensorDataAsJson*. Nakon što su podaci obrađeni, koristi se emitiranje signala naredbe `emit temperatureUpdated` i `emit humidityUpdated` kako bi se obavijestili ostali dijelovi aplikacije o novim vrijednostima temperature i vlažnosti. Također, koriste se *QReadLocker* i *QWriteLocker* unutar bloka koda koji osigurava siguran pristup dijeljenom resursu, odnosno čitanja i pisanja iz datoteke, u višedretvenom okruženju. Ovo je vrlo važno jer drugi dijelovi kôda će pokušavati čitati tu datoteku kako bi se prikazali podaci.

6.3.3.2. Glavna progamska petlja

Isječak kôda 7 prikazuje glavnu programsku petlju aplikacije, koja uključuje inicijalizaciju aplikacije, postavljanje glavnog prozora, konfiguraciju DHT22 senzora, kao i postavljanje veze između signala emitiranih iz DHT22 klase i ažuriranja korisničkog sučelja.

U ovom dijelu koda, `fileMutex` je instanca `QReadWriteLock` koja se koristi kao za-

```cpp
#include
#include
#include
#include
#include "mainwindow.h"
#include "dht22.h"
#include
#include
#include
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    try {
        QReadWriteLock fileMutex;

        // MainWindow setup

        QString fileName = "sensorDataDHT22.json";
        MainWindow w(&fileMutex, fileName);

        // QQuickWidget setup for QML
        QQuickWidget *qmlWidget = new QQuickWidget;
        qmlWidget->setSource(QUrl(QStringLiteral("qrc:/HuPiTp/Main.qml")));
        qmlWidget->setResizeMode(QQuickWidget::SizeRootObjectToView);
        w.setCentralWidget(qmlWidget);

        // DHT22 setup
        DHT22 *dht22 = new DHT22(&fileMutex); // Create DHT22 object on the stack

        auto readDataFunc = [=]() {
            dht22->readAndOutputSensorDataAsJson(fileName);
        };

        // Use QtConcurrent::run with the lambda function
        QFuture future = QtConcurrent::run(readDataFunc);

        // Connect signals to update both QWidget and QML interfaces
        QObject::connect(dht22, &DHT22::temperatureUpdated, &w, [&](float celsius,
        ↪  float fahrenheit) {
            w.updateTemperature(celsius, fahrenheit);
            if (qmlWidget->rootObject()) {
                qmlWidget->rootObject()->setProperty("temperatureValue", celsius);
                qmlWidget->rootObject()->setProperty("temperatureFValue",
                ↪  fahrenheit);
            }
        });

        QObject::connect(dht22, &DHT22::humidityUpdated, &w, [&](float humidity) {
            w.updateHumidity(humidity);
            if (qmlWidget->rootObject()) {
                qmlWidget->rootObject()->setProperty("humidityValue", humidity);
            }
        });
        w.show();
        return app.exec();
    } catch (const std::exception &e) {
        QMessageBox::critical(nullptr, "Error", QString("An error occurred in
        ↪  main.cpp: %1").arg(e.what()));
        return -1;
    }
}
```

Isječak koda 7: Implementacija glavne petlje aplikacije

jednički mutex za zaključavanje dijeljenog resursa, što omogućuje siguran pristup datoteci koja sadrži podatke senzora.

`QObject::connect()` funkcija se koristi za povezivanje signala iz `DHT22` klase s odgovarajućim slotovima u `MainWindow` klasi. Kada se signal `temperatureUpdated` ili `humidityUpdated` emitira iz instance `DHT22`, povezani slotovi u `MainWindow-u` se automatski pokreću. Ovi slotovi zatim ažuriraju korisničko sučelje aplikacije s novim podacima temperature i vlažnosti. Također, ako je QML korisničko sučelje aktivno putem `QQuickWidget`-a, svojstva `temperatureValue` i `humidityValue` se postavljaju na odgovarajuće vrijednosti kako bi se ažurirao prikaz na korisničkom sučelju.

6.3.3.3. Grafičko sučelje - QML objekti

U ovom dijelu aplikacije koristi se QML (Qt Modeling Language), deklarativni jezik za opisivanje korisničkih sučelja u Qt okruženju. Glavna svrha QML-a je olakšati kreiranje interaktivnih korisničkih sučelja. U implementiranoj aplikaciji, QML se koristi za definiranje prikaza temperature, vlažnosti i loga. Isječak kôda 8 prikazuje `Main.qml` datoteku.

QML kôd deklarativno definira korisničko sučelje koje sadrži tri korisničko definirana elementa: `Temperature`, `Humidity` i `Log`.Ovi elementi se dinamički ažuriraju na temelju promjena temperatura i vlažnosti. Svaki od ovih elemenata predstavljen je odvojenim QML datotekama radi bolje organizacije koda i modularnosti aplikacije.

Nadalje pod glavnim deklarativnim elementom `item` definirana su tri varijable svojstva: `temperatureValue`, `temperatureFValue` i `humidityValue` koje se koriste za praćenje trenutnih vrijednosti temperature i vlažnosti. Kada se ove vrijednosti promijene, odgovarajući QML elementi se automatski ažuriraju kako bi odražavali nove podatke. Osim toga, za prikaz grafičkih podataka koristi se `mainwindow.h`, koji pruža metode za prikaz dnevnih, mjesečnih i statistike uživo.

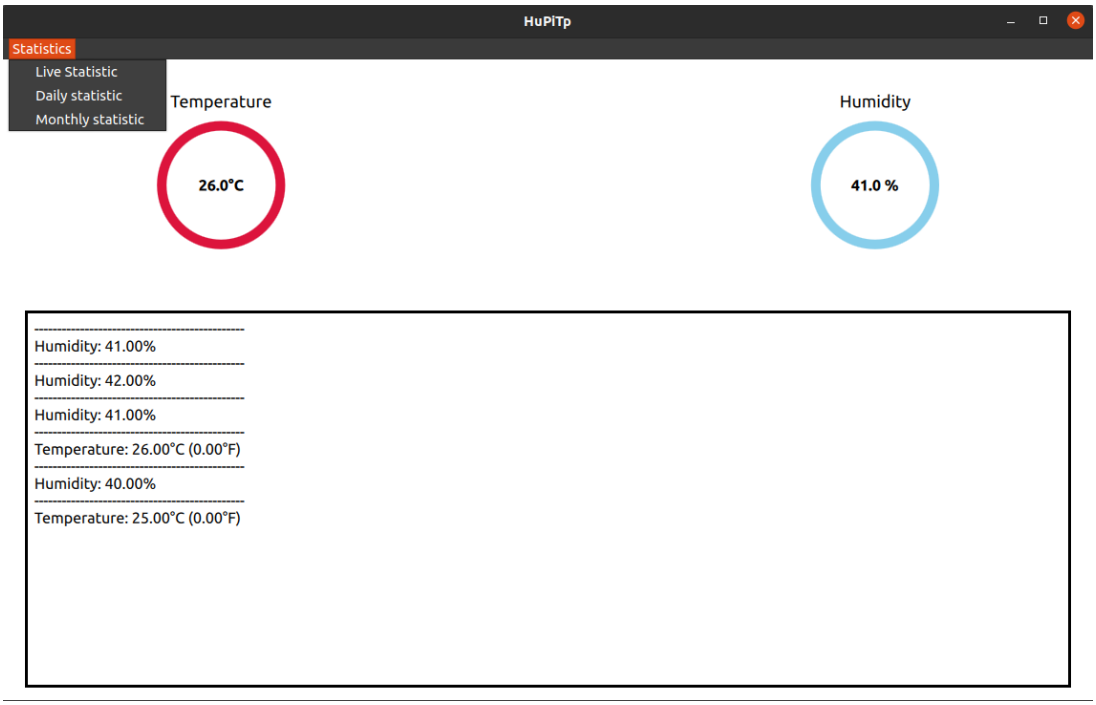
6.3.3.4. Kratak pregled rada aplikacije

Slika 20 prikazuje glavni prozor grafičkog sučelja HuPiTp aplikacije. Kao što je definirano u isječku kôda 8, sučelje se sastoji od elementa `Temperature`, `Humidity` i `Log`. Temperatura je označena crvenom kružnicom te prikazuje zadnje (trenutno) očitanje temperature. S druge strane vlažnost je označena svijetlo plavom bojom te ima istu funkcionalnost. Log je ispod uokviren obrubom crnom pravokutnika te prikazuje zadnje promjene. To znači da ako je očitanje temperature ili vlažnosti bilo identično, to ne će biti postavljeno u log, iako će podaci biti zabilježeni u *.json* datoteci. To je razlog zašto vidimo više uzastopnih očitanja vlažnosti. U gornjem lijevu kutu vidljiva je traka izbornika s padajućim izbornikom statistike. Tu se nude tri opcije grafičkog prikaza podataka. Svaki od izbornika otvara novi prozor za pregled podataka:

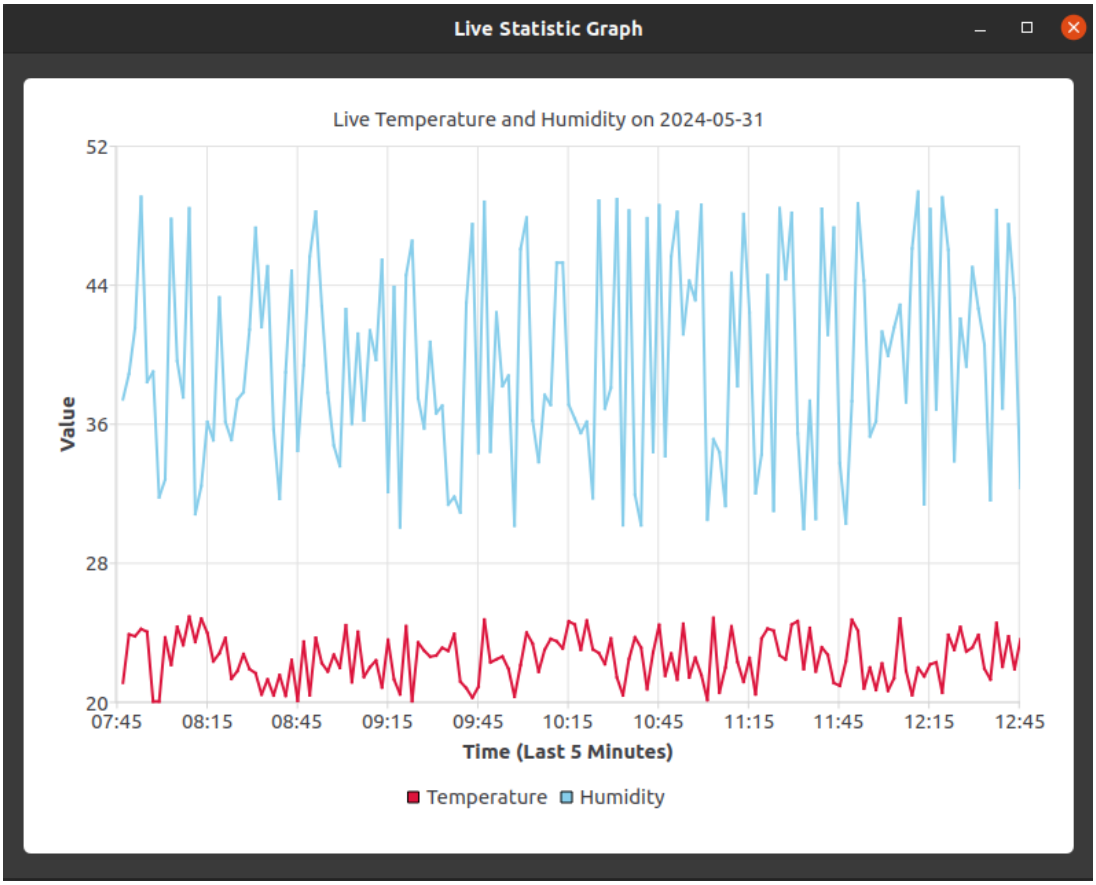
```
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15
import QtQuick.Controls.Material 2.15

Item {
visible: true
anchors.fill: parent
property real temperatureValue: 0
property real temperatureFValue: 0
property real humidityValue: 0

onHeightChanged: {
    humidityObj.updateCircle(height)
    temperatureObj.updateCircle(height)
}

Temperature {
    id: temperatureObj
    temperature: temperatureValue // Pass temperatureValue to Temperature
    anchors.left: parent.left
    anchors.top: parent.top
    anchors.leftMargin: 1/5 * parent.width // Add space from the left
    anchors.topMargin: 1/20 * parent.height  // Add space from the top
}

Humidity {
    id: humidityObj
    humidity: humidityValue
    anchors.right: parent.right
    anchors.top: parent.top
    anchors.rightMargin: 1/5 * parent.width   // Add space from the right
    anchors.topMargin: 1/20 * parent.height// Add space from the top
}

Log {
    id: log
    anchors {
        bottom: parent.bottom
        left: parent.left
        right: parent.right
        topMargin: 50
    }
    width:parent.width
}

Component.onCompleted: {

}

onTemperatureValueChanged: {
    temperatureObj.updateTemperature(temperatureValue)
    log.updateTemperatureLog(temperatureValue,temperatureFValue)
}

onHumidityValueChanged:{
    humidityObj.updateHumidity(humidityValue)
    log.updateHumidityLog(humidityValue)
}
}
```

Isječak koda 8: Implementacija grafičkog sučelja pomoću QML-a

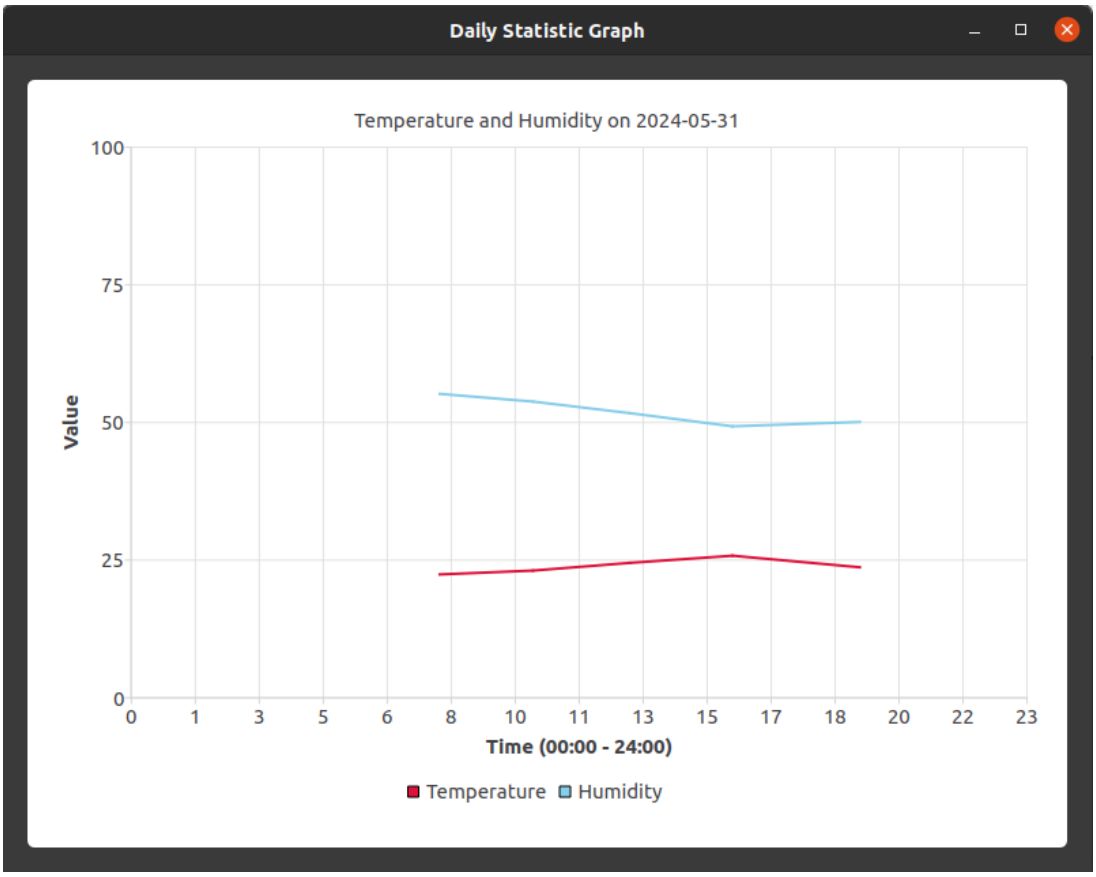
1. Aktualna učitavanja - linijski graf koji prikazuje očitavanja temperature i vlažnosti u proteklih pet minuta. Ovaj graf se uživo ažurira (Slika 21).
2. Dnevna statistika - linijski graf koji prikazuje cjelodnevno kretanje temperature i vlažnosti. Ovaj graf je statički, odnosno prilikom pokretanja generira se graf na temelju zadnjih podataka za trenutni dan, ali nakon što je graf prikazan ne ažurira se (Slika 22).
3. Mjsečena statistika - graf barova koji prikazuje prosječno stanje vlažnosti i temperature za tekući mjesec po danu. Ovaj graf je također stacionaran (Slika 23).



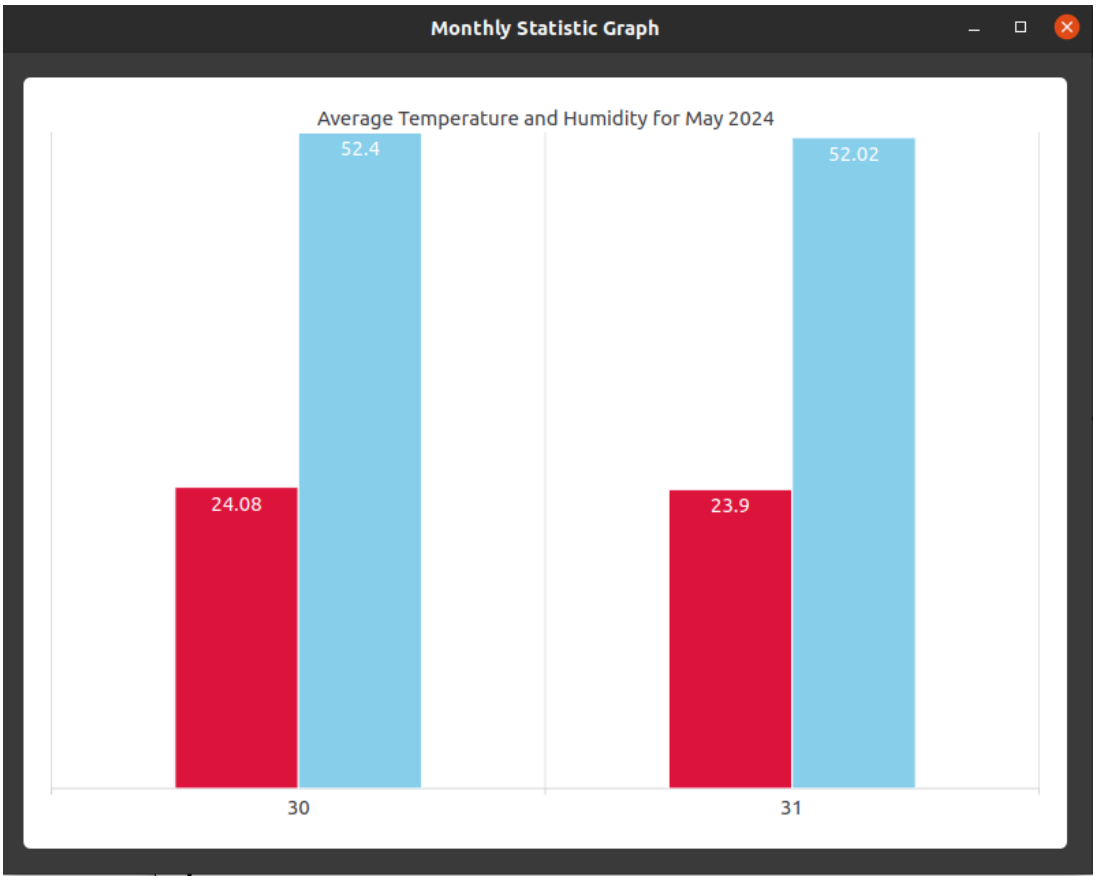
Slika 20: Prikaz glavnog prozora grafičke aplikacije HuPiTp



Slika 21: Prikaz prozora aktualne statistike



Slika 22: Prikaz prozora dnevne statistike



Slika 23: Prikaz prozora mjesečne statistike

7. Zaključak

U ovom radu definirani su koncepti ugradbenih sustava i interneta stvari (IoT), istraženi su različiti pristupi razvoju softverskih rješenja za ugradbene uređaje i IoT uređaje, te analizirana suvremena razvojna okruženja. Fokus je bio na usporedbi Buildroota i Yocto projekta, pružajući detaljan pregled njihovih razvojnih procesa i preporučenih praksi. Kroz praktičnu implementaciju, stvorena je prilagođena slika operacijskog sustava za Raspberry Pi koristeći Yocto projekt, uključujući i grafičku aplikaciju za praćenje temperature i vlažnosti s pomoću DHT22 senzora. Ovaj proces razvoja zahtijevao je mnogo iteracija, od rada na konfiguracijama i prilagodbama do rješavanja grešaka i ovisnosti. Međutim, kroz taj napor, ostvareni su zadovoljavajući rezultati, demonstrirajući funkcionalnosti i mogućnosti koje nude suvremena razvojna okruženja za ugradbene sustave i IoT uređaje. Integracija grafičkog sučelja s pomoću QML-a dodatno je poboljšala korisničko iskustvo i pokazala snagu deklarativnog pristupa u izradi jednostavnijih interaktivnih korisničkih sučelja. Kroz ovo istraživanje, steklo se razumijevanje procesa razvoja ugradbenih sustava i IoT uređaja, kao i praktično iskustvo u radu s modernim razvojnim alatima i okruženjima. S obzirom na brzi razvoj tehnologije u ovom području, ovaj rad pruža čvrstu osnovu za potencijalna daljnja istraživanja i razvoj u području ugradbenih sustava i IoT tehnologija.

Popis literature

[1] L. De Micco, F. L. Vargas i P. I. Fierens, „A Literature Review on Embedded Systems," *IEEE Latin America Transactions*, sv. 18, br. 02, str. 188–205, 2020. DOI: 10.1109/TLA.2020.9085271.

[2] Y. Hajjaji, W. Boulila, I. R. Farah, I. Romdhani i A. Hussain, „Big data and IoT-based applications in smart environments: A systematic review," *Computer Science Review*, sv. 39, str. 100 318, 2021., ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2020.100318.

[3] A. Banafa, „6 Three Major Challenges Facing IoT," *Secure and Smart Internet of Things (IoT): Using Blockchain and AI*. 2018., str. 33–44.

[4] M. Fahmideh, A. Ahmad, A. Behnaz, J. Grundy i W. Susilo, „Software Engineering for Internet of Things: The Practitioners' Perspective," *IEEE Transactions on Software Engineering*, sv. 48, br. 8, str. 2857–2878, 2022. DOI: 10.1109/TSE.2021.3070692.

[5] Vocal.com. „Software Components of Embedded Linux Systems." (n.d.), adresa: https://vocal.com/software-modules/embedded-software-modules/software-components-of-embedded-linux-systems/ (pogledano 28. 3. 2024.).

[6] W. Almesberger, „Booting linux: The history and the future," *Proceedings of the Ottawa Linux Symposium*, 2000.

[7] R. KUMAR. „Linux Tutorials: root file systems in linux." (2022.), adresa: https://www.devopsschool.com/blog/linux-tutorials-root-file-systems-in-linux/#respond (pogledano 28. 3. 2024.).

[8] „Filesystem Hierarchy Standard." Edited by Rusty Russell, Daniel Quinlan, Christopher Yeoh. (), adresa: https://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.pdf.

[9] Linux Foundation. „Linux Foundation Reference Specifications." Pristupljeno: DD. MM. YYYY. (n.d.), adresa: https://refspecs.linuxfoundation.org/.

[10] B. Project, *Buildroot User Manual*, Dostupno: 2024-05-21, 2024.

[11] M. Brambilla, J. Cabot i M. Wimmer, *Model-driven software engineering in practice* (Synthesis Lectures on Software Engineering 1). Morgan & Claypool Publishers, 2012., sv. 1, str. 1–182.

[12] O. Kautz, A. Roth i B. Rumpe, *Achievements, Failures, and the Future of Model-Based Software Engineering.* 2018.

[13] S. U. Khan, A. W. Khan, F. Khan, M. A. Khan i T. K. Whangbo, „Critical Success Factors of Component-Based Software Outsourcing Development From Vendors' Perspective: A Systematic Literature Review," *IEEE Access*, sv. 10, str. 1650–1658, 2022. DOI: 10.1109/ACCESS.2021.3138775.

[14] A. Bucaioni, F. Ciccozzi, A. Di Salle i M. Sjödin, „From low-level programming to full-fledged industrial model-based development: the story of the Rubus Component Model," *Software and Systems Modeling*, sv. 22, br. 4, str. 1085–1097, 2023., Cited by: 1; All Open Access, Hybrid Gold Open Access. DOI: 10.1007/s10270-023-01107-3.

[15] D. Bacciu, K. Tserpes, M. Coppola i dr., „TEACHING: A Computing Toolkit for Building Efficient Autonomous appliCations Leveraging Humanistic INtelliGence," *Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge*, serija FRAME '23, Orlando, FL, USA: Association for Computing Machinery, 2023., str. 37–39, ISBN: 9798400701641. DOI: 10.1145/3589010.3594886.

[16] A. S. Tanenbaum, *Modern Operating Systems*, 3rd. Upper Saddle River, NJ: Pearson/-Prentice Hall, 2008., str. 160, ISBN: 978-0-13-600663-3.

[17] P. Hambarde, R. Varma i S. Jha, „The Survey of Real Time Operating System: RTOS," *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, 2014., str. 34–39. DOI: 10.1109/ICESC.2014.15.

[18] A. Leppäkoski, E. Salminen i T. D. Hämäläinen, „Framework for industrial embedded system product development and management," *2013 International Symposium on System on Chip (SoC)*, 2013., str. 1–6. DOI: 10.1109/ISSoC.2013.6675265.

[19] R. Delgado, J. Park i B. W. Choi, „Open embedded real-time controllers for industrial distributed control systems," *Electronics*, sv. 8, br. 2, str. 223, 2019.

[20] B. Project. „Buildroot Download Page." Dostupno: 2024-05-21. (2024.), adresa: https://buildroot.org/download.html.

[21] V. Reznikov, „Creating tailored OS images for embedded systems using Buildroot, Criação de SO customizáveis para um ambiente embebido utilizando o Buildroot," Master thesis, Universidade do Minho, Braga, Portugal, 2019.

[22] T. Y. Project. „Yocto Project," The Yocto Project. (2024.), adresa: https://www.yoctoproject.org/ (pogledano 27. 5. 2024.).

[23] Bootlin. „Yocto Slides." (2024.), adresa: https://bootlin.com/doc/training/yocto/yocto-slides.pdf (pogledano 18. 6. 2024.).

[24] T. Perä, „Comparison of custom embedded Linux build systems: Yocto and Buildroot," en, Master's thesis, Aalto University, 6. 2022.

[25] Gfilinic, *meta-pisense GitHub Repository*, https://github.com/Gfilinic/meta-pisense.git, 2024.

[26] WiringPi, *WiringPI GitHub Repository*, https://github.com/WiringPi/WiringPi.git, 2024.

[27] W. A. Goran Filinić, *WiringPI CMake branch - GitHub Repository*, https://github.com/WiringPi/WiringPi/tree/cmake, 2024.

[28] SparkFun Electronics. „DHT22 Temperature/Humidity Sensor Datasheet.” Dostupo: 2024-05-30. (2011.), adresa: https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf.

[29] Pinout.xyz. „Raspberry Pi GPIO Pinout.” Dostupno: 2024-05-30. (2024.), adresa: https://pinout.xyz/.

[30] Gfilinic, *HuPiTp GitHub Repository*, https://github.com/Gfilinic/HuPiTp.git, 2024.

Popis slika

Popis tablica

Popis isječaka koda