CHALMERS
UNIVERSITY OF TECHNOLOGY

Minnessäkra applikationer för mikroprocessor

Examensarbete inom Data- och Informationsteknik

Isak Einler Larsson
Gabriel Lindeby

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2018

Examensarbete 2018

Minnessäkra applikationer för mikroprocessorer

Isak Einler Larsson
Gabriel Lindeby

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet
Göteborg, Sverige 2018

Minnessäkra applikationer för mikroprocessorer
Isak Einler Larsson
Gabriel Lindeby

Handledare: Roger, Johansson
Rådgivare: Magnus Persson, Raybased
Examinator: Peter Lundin, Institutionen för data- och informationsteknik

Examensarbete 2018
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslag: En bild på en R-Puck. Den möjliggör mätning och styrning av elkomponenter i fastigheter.

Typeset in LaTeX
Göteborg, Sverige 2018

Minnessäkra applikationer för microprocessorer
Isak Einler Larsson
Gabriel Lindeby
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet

Sammanfattning

Språket C har många fördelar och det är vanlig att det förekommer i mjukvara för inbäddade system. Språket ger utvecklaren full tillgång till hårdvara och hantering av minne görs manuellt. Fördelarna med C är att applikationerna oftast blir minnessnåla och effektiva jämnfört med andra högnivåspråk. C har också stora säkerhetsbrister, exempelvis buffer-överflöde, minnesläckor, oinitierat minne och användning av noll-pekare. Den här rapporten kommer att jämföra alternativa strategier för utveckling av mjukvara för cortex m0+ med säkrare minneshantering än traditionell C-programmering.

Nyckelord: Minnesäkerhet, Rust, Java-till-C översättning, Mikroprocessor, JCGO

Abstract

The language C is great when it comes to writing software for embedded systems. You have full access to hardware functionality and memory management. Software written in C also has the benefit of very low to zero overhead compared to languages at a higher level of abstraction. However the freedom of C comes with several safety disadvantages such as buffer overflows, memory over read, memory leaks, uninitialized memory and accessing null pointer. This report will compare alternative ways of writing software especially aimed for the ARM cortex m0+ with safety in mind.

Keywords: Memory safety, Rust, Java-to-C translation, Microprocessor, JCGO

Förord

Vi vill tack våra rådgivare på Raybased, Magnus Persson och Jan Ryderstam, som gav oss möjligheten att få arbeta med detta projekt men även för all hjälp och stöd vi fått. Vi vill även tacka Ivan Maidanski som utvecklat JCGO och som mer än gärna svarande på frågor. Sist men inte minst vill vi tacka Roger Johansson som har varit en bra handledare.

Isak Einler Larsson, Gabriel Lindeby, Göteborg, Juni 2018

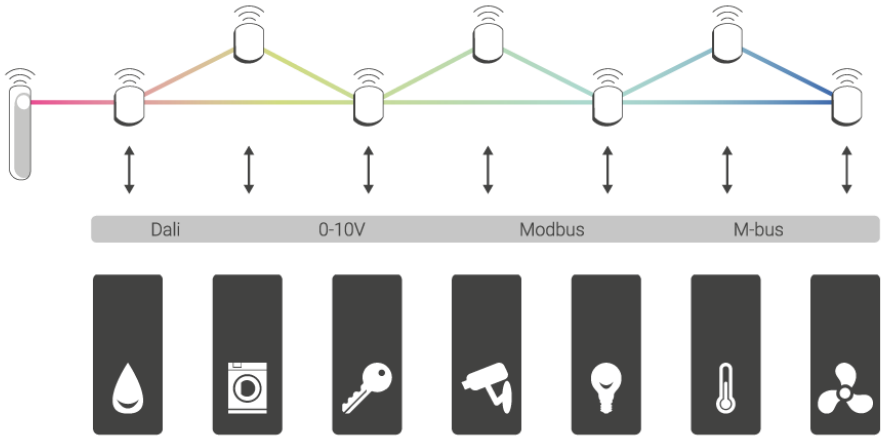
Sammanställning av förkortningar

JVM	Java Virtual Machine
JNI	Java Native Interface
FreeRTOS	Free Real Time Operating System
IoT	Internet of Things
J2C	Java to C
FFI	Foreign Function Interface
GDB	GNU Debugger

Innehållsförteckning

1
Introduktion

Raybased är ett företag som främst riktar sig mot att integrera modern teknik i äldre befintliga fastigheter och göra dem *smarta* eller *uppkopplade*. Dessa fastigheter använder äldre teknik och är därför i behov av bl.a energieffektivisering. Problemet som tillkommer är att det är svårt uppskatta den totala kostnaden. Raybased har tagit fram en lösning som kombinerar konsistens och prestanda hos ett trådbundet system med pris och användarvänliga aspekter av ett trådlöst system. Det kan göras genom att installera olika moduler som sensorer, reläer och radiosändare som styrs av en tillhörande mikroprocessor. Detta gör det möjligt att styra och optimera flera olika enheter i en fastighet; Värme, ventilation och belysning och därmed bidra till kostnads och energieffektivisering.



Figur 1.1 - En systemstruktur för Raybaseds produkt. En integrerande plattform som baserar sig på samtliga enheter.

1.1 Bakgrund

För att öka värdet av sin produkt och vara en attraktiv konkurrent på marknaden vill Raybased hitta en lösning där deras kunder själva kan ändra eller vidareutveckla systemets programvara. Programvaran som körs på mikroprocessorerna är skrivet i språket C, vilket är vanligt för programmering på låg nivå. Detta beror på att C ger enkel tillgång till hårdvaru- och minnesportarna. Koden sammanställs till maskinkod som processorn förstår utan en virtuell maskin däremellan. Problemet med att låta användaren skriva C-kod som direkt utförs av mikroprocessorn är att det introducerar risken att manipulera den befintliga programvaran, vilket kan resultera i säkerhetsproblem. För att undvika detta krävs restriktioner för att säk-

ertställa att programmet inte utför några riskfyllda operationer som kan få systemet att krascha eller att utnyttja friheten med C för att manipulera övrig programkod.

1.2 Syfte

Syftet med projektet är att utforska säkrare alternativ till traditionell mjukvaruutveckling i C och jämnföra dess för- och nackdelar. Med hjälp av de slutsatser som vi drar ska vi därefter ta fram ett koncept för utveckling av applikationer till Raybaseds plattform. Det kommer i sin tur att möjliggöra utveckling av tredjepartsapplikationer utan att äventyra plattformens stabilitet och säkerhet.

1.3 Mål

Målet med projektet är att ta fram ett koncept för en plattform eller flöde baserat på säkrare, befintliga alternativ än C: Använda maskinorienterade språk som t.ex Rust, använda lättviktig Java Virtual Machine(JVM) eller översätta mellan ett godtyckligt säkrare språk till C. Ett krav för att metoden ska vara användbar är att den ska fungera med det befintliga systemet. Konceptet är sedan tänkt att utvecklas för tredjepartsföretag och Raybased.

1.4 Avgränsning

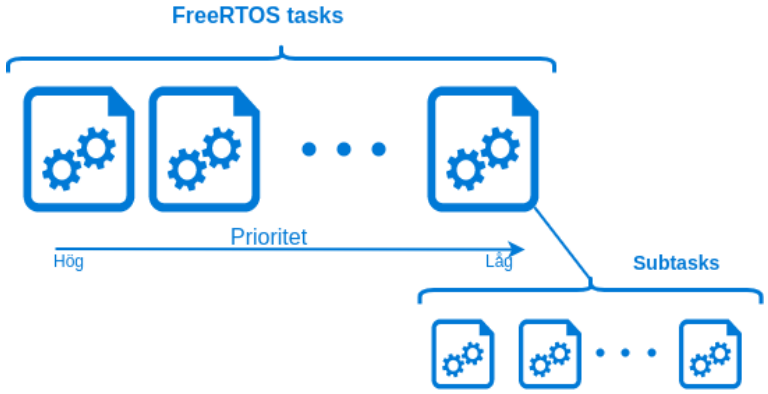
Vi kommer återanvända så mycket som är möjligt av det befintliga systemet. Översättning mellan språk kommer att utföras av befintligt tjänst och kommer ej utveckla nya verktyg.

2
Teknisk Bakgrund

I detta kapitel förklaras några av de begrepp som används i rapporten.

Free Real Time Operating System, (FreeRTOS) är ett minimalt realtidsinriktat operativsystem som vanligen används i mindre inbäddade system. Mindre operativsystem idag tillåter *multitasking* vilket innebär möjligheten att köra fler program samtidigt. Men i verkligheten kan varje processorkärna endast tillåta en enda tråd att köra oavsett tidpunkt. I ett operativsystem finns en schemaläggare som ansvarar över vilket program som ska köra och när det ska köra men ger visionen att flera program körs samtidigt. Detta görs genom att operativsystemet snabbt byter mellan de olika programmen. Detta kallas pseudoparallellism. FreeRTOS fungerar på liknande sätt men uppnår detta genom att användaren anger en prioritetslista för olika exekveringar.[4]

I Raybaseds implementation av FreeRTOS finns möjligheten att skapa tasks med olika prioritet. Varje task har en egen meddelandekö och hanterarmetod. Vid skapandet av en ny task bestäms också vilken prioritet den ska ha. I Raybaseds fall är exempelvis tasken för hantering av radiokommunikation extra kritisk och därför högre prioriterad än något annat. Det vanligaste är dock att en task inte har några kritiska realtidskrav. Då finns möjligheten att skapa en subtask till en parenttask. Fördelen med det är att alla subtasks delar på samma meddelandekö vilket minskar storleken på använt minne. Parenttaskens uppgift blir då att fördela meddelanden från kön och exekveringstid mellan dess subtaskar.



Figur 2.1 - Visar hur en parenttask kan ha flera subtasks

För att tasks eller block ska kunna kommunicera med varandra så skickar de meddelanden. Dessa meddelanden skickas inte bara mellan block i en puck utan kan

också skickas till block i en avlägsen puck via radiokommunikation.[1] Denna typ av meddelandehantering gör systemet skalbart för stora konfigurationer. Varje meddelande innehåller ett id till mottagande task samt en array med data.

En ***Java Virtual Machine(JVM)*** är en abstrakt maskin eller virtuell maskin, alltså inte en befintlig, fysisk dator. Denna virtuella maskin drivs på ett system och ger en möjligheten att köra Java-program. En JVM fungerar så att den tolkar så kallad bytekod som skapas vid kompilering av javakod vilket gör Java till ett aktuellt språk att utveckla programvara i eftersom det är plattformsoberoende.[10]

"Sakernas Internet" / Internet of Things eller IoT är ett nätverk av uppkopplade, fysiska vardagsföremål som ofta kommunicerar med varandra via internet. Typiska IoT-produkter kan vara hushållsapparater, fordon, mobiler, byggnader och kläder. Det som skiljer de olika produkterna är att de alla har sitt egna inbyggda datorsystem och kan därför unikt identifierat kommunicera inom internets infrastruktur.

En ***kompilator*** är ett program som översätter kod från ett godtyckligt språk, källspråket, till ett annat språk, målspråket. Där källspråket är språket som användaren har skrivit sin kod i och målspråket är språket en dator kan läsa. Med andra ord är en kompilator en typ av översättare som gör koden läsbart för digitala enheter.[5]. I denna rapport kommer ordet *ahead of time* att nämnas. *Ahead of time* innebär att alla kodgenereringar och optimeringar händer innan programmet körs.

Det finns även ***kors-kompilator*** vilket innebär att en kompilator kan översätta kod till andra plattformar som kompilatorn inte befinner sig på. Ett exempel hade varit en kompilator som kompilerar C-kod för en godtycklig ARM-processor vilket bygger på en annan arkitektur än en vanlig dator. En kors-kompilator är nödvändigt för ett system ska kunna kompilera kod för flera olika plattformar och inte bara till sin egen plattform, vilket bidrar till flexibilitet.

Öppen källkod / Open source innebär att det är något som vem som helst kan modifiera, vidareutveckla och dela med sig av, då dess design är offentligt tillgänglig för alla. Öppen källkod ger t.ex möjligheten att enkelt kunna göra utbyten, samverkningar, samhällsorienterad utveckling och lösningar för att ta fram snabba prototyper.

JCGO(uttalas "j-c-go") och är en mjukvara som ger möjligheten att översätta kod från Java till språket C. C-koden som genereras är av god kvalitet och kan användas genom att kompileras mot godtycklig plattform för att sedan köras.

Rust är ett programmeringsspråk som är riktat mot lågnivåsystem. Dess filosofi är säkerhet och effektivitet. Jämfört med andra språk som använder sig av pekare eller skräpsamling så använder Rust något som kallas *Ownership*.[2]

[1] Den centrala enheten i systemet kallas puck och är byggd på en mikroprocessor.
[2] Ownership beskrivs tydligare i 4.2

Garbage collector eller skräpsamlare är ett system som dynamiskt hanterar minne i Java. Målet är att plocka upp alla resurser och arbetsminne som inte längre kommer användas och återvinna det för ny användning.

JUnit är ett ramverk baserat på öppen källkod och är ett testramverk designat för att enkelt kunna köra tester mot javakod. "Unit"-testning innebär att man bryter ner sitt program i mindre delar, därav namnet och sedan kör tester på de olika delarna. Tester körs ofta periodiskt och ofta direkt efter att ytterligare funktionalitet lagts till, en mer formell benämning för detta är "regressions"-testning. Ju oftare tester körs ju lättare är det att hitta eventuella problem.[7]

Overhead kan vara kombinationer av indirekt beräkningstid, minne eller andra resurser som krävs för att utföra nödvändiga operationer.

Puck är samlingsnamnet på produkterna framtagna av Raybased. En puck består av en mikroprocessor och hanterar till exempel elkomponenter eller sensordata.

3
Metod

3.1 Datainsamling

Projektet tog sin början med en studie där planen var att se över de olika alternativen som fanns. Det gjordes främst genom befintliga rapporter men även via testning som vi själva utfört. Chalmers bibliotek hade även en del intressant information som vi kunde använda, även äldre kandidat arbeten har varit givande att läsa. Sen har även diskussioner förts med våra handledare för att undvika eventuella fallgropar. I den omvärdesanalys som har genomförts av liknade projekt har rapporten *A Rust-based Runtime for the Internet of Things*[3] varit intressant. Den förklarar hur Rust är uppbyggt och gav god återkoppling på hur Rust kan tillämpas på mikroprocessorer. En stor skillnad mellan problemets som beskrivs i den rapporten och vårt problem är att Raybased har en stor kodbas skriven i C som ska bevaras. Vidare undersöktes *Code Generation and Optimization for Java-to-C Compilers*[18] som gav bakgrund till översättning mellan java och C. Detta bidrog till en bra start och nya perspektiv som kunde tänkas användas för utveckling av detta projekt.

3.2 Arbetsmetod

Projektet är utformad efter en studie där man gjort en omvärldesanalys och tagit fram information som kan vara lämplig. Även samråd med handledarna tog plats vilket gav ytterligare information och grund till projektet. Utifrån detta togs tre olika strategier fram. Arbetsmetoden blev att undersöka och testa alternativen och ta fram alla för- och nackdelar och sedan jämföra de med varandra. Målet var att implementera denna lösning mot systemet Raybased utvecklat. Med andra ord baserar sig arbetesmetoden på en undersökning och utvärdering av tre möjliga tillvägagångsätt.

4

Genomförande och implementation

För att lösa problemet har vi valt att undersöka tre olika typer av strategier. Vi kommer att undersöka möjligheterna att använda ett högnivåspråk, interpretator och lågnivåspråk med stöd för säker minneshantering och översättning från ett högnivåspråk till C-kod. För att lösningen ska vara användbar så måste det gå att integrera den i det befintliga systemet som använder sig av FreeRTOS. I detta kapitlet kommer vi att redogöra för de olika strategierna och visa på en möjlig implementation.

4.1 JVM

Genom att använda en Java Virtual Machine skapas möjligheten att köra javakod på nästan vilket godtyckligt system som helst. Java är idag ett av de största språken och majoriteten av programmerare behärskar det, vilket vidgar målgruppen för systemet.[11] Utöver det så bidrar Java även till säkrare kod då den, till skillnad från C och C++, använder sig av en *Skräp samlare*. Dess funktion är att automatiskt och dynamiskt hantera minne genom att hela tiden försöka återvinna skräp. Med skräp menas arbetsminne eller resurser som tilldelats olika objekt, men som inte använts eller kommer att användas igen. Genom att samla på sig allt skräp kan den frigöra minne som istället kan återanvändas till något annat.[9].

Det finns dock nackdelar med att köra en JVM. Eftersom koden måste köras på en virtuell maskin innebär det att den kommer att köras på högre nivå än maskinorienterade språk och brukar därför vara långsammare. Hur stor skillnaden är beror givetvis på systemets prestanda. Detta beror på att Java är ett system-neutralt språk och kan inte optimeras för specifik hårdvara.[6]

Tanken är att använda en avskalad JVM som kräver minimalt med processorkraft och minne. På så sätt går det att skriva kod i ett säkrare språk och ändå bibehålla tillräckligt med prestanda för att driva det i realtid.

4.2 Rust

Rust är ett minnessäkert lågnivåspråk som ger möjligheten till effektivitet samtidigt som den bevarar säkerhet. Dess filosofi är att minska risker för osäker mjukvara och

istället sätta en högre standard hos programmerare och företag. Rust är även avsett för större, parallella system som kräver högre säkerhet och underhåll.

Istället för att använda pekare som C och C++ gör så använder sig Rust av *Ownership* vilket är ett antal regler som måste uppfyllas för att uppnå minnessäkerhet. Övergripande så innebär reglerna att en variabel alltid måste ha en ägare som har tillåtelse att modifiera variabeln. Det finns alltid exakt en ägare av en variabel och om ägaren inte finns i den aktuella kontexten så är variabeln inte giltig längre. Genom att kolla dessa regler under kompilering kan minnesäkerhet garanteras och bidrar inte till någon *overhead* vid exekvering av programmet[15].

Tanken är att Rust och C-kod ska kunna samexistera för att inte behöva skriva om kod i det befintliga systemet. Detta är möjligt genom att skapa säkra gränssnitt till metoder som är implementerade i C med Foreign Function Interface (FFI)[12]. Det går också att kalla på metoder implementerade i Rust från C[2].

Rust består av två stycken språk safe och unsafe Rust. Skillnaden mellan safe och unsafe är att unsafe inte behöver förhålla sig till de regler som garanterar minnessäkerhet och därmed gör språket osäkert[13]. I vissa fall är det nödvändigt att använda unsafe Rust exempelvis vid användning av FFI eftersom det kallar på C kod där minnsesäkerhet inte kan garanteras. Utvecklarna av nya komponenter kommer endast ha möjlighet att skriva safe Rust.

För tillfället finns ingen officiell binär för korskompilering till Cortex-M0+[14]. Det finns heller inget stöd för den arkitekturen i Cargo som bland annat används för att hantera beroende till andra paket och för att underlätta vid kompilering av ett projekt. För att lösa problemet med att det inte finns något enkelt sätt att kompilera för denna arkitekturen så kan det inofficiella opensource projektet Xargo[1] användas som ersättare till Cargo.

4.3 JCGO

Det sista alternativet är att översätta från Java till C-kod med hjälp av det öppna källkodsprogrammet JCGO. JCGO är en *ahead of time* kompilator som översätter hela javaprogram till plattformsoberoende C-kod. Därefter kan en fristående korskompilator användas för generering av optimerad maskinkod till den specifika processorn.

JCGO översätter direkt ifrån källkod utan att först kompilera till javabinär. Det gör det möjligt att generera C-kod som är läsbar. Tanken är dock inte att den genererade koden ska ändras och underhållas. Även om den genererade koden är läsbar så är det svårt att återskapa Java-koden vilket kan vara positivt om källkoden inte ska vara publik. En konsekvens av att JCGO översätter direkt i från källkod är att källkod till eventuella bibliotek också måste finnas tillgänglig.

Eftersom JCGO använder sig av *ahead of time*-kompilering så finns möjlighet till att prioritera minne över hastighet och tvärtom. Det är också möjligt att bara kompilera de delar av Javas virtuella maskin som används vilket sparar minne. Precis som fördelarna med att använda en JVM så minskar utvecklarens möjlighet att manipulera specifika platser i minnet. Det finns också större möjlighet till att testa kod vilket bidrar till säkrare applikationer. JCGO underhålls inte längre och har stöd upp till javaversion 1.6.

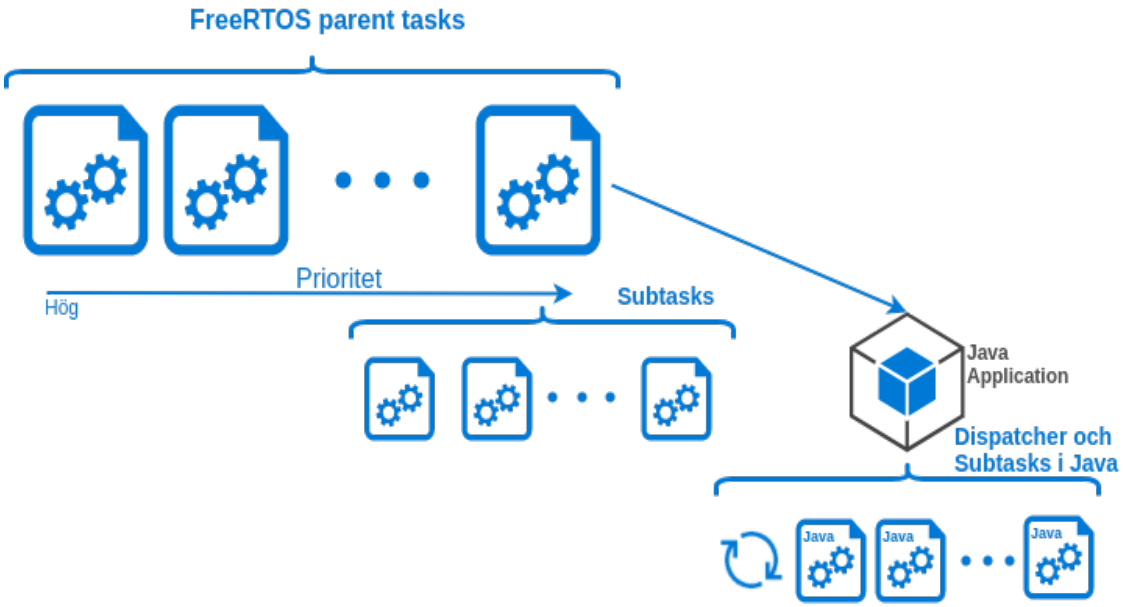
Tanken är att fortfarande använda den befintliga C-koden och implementera nya moduler i Java. De nya modulerna kommer att fungera som en separat applikation som FreeRTOS startar upp. För att de nya modulerna ska komma åt funktioner som är implementerade i C så kommer Java Native Interface(JNI) att användas vilket är ett sätt att kalla på bibliotek som är implementerade i andra språk än Java[8]

4.4 Implementation

Efter att ha studerat de olika möjliga strategierna så valde vi att fortsätta med översättning av ett högnivåspråk till C-kod med hjälp av JCGO. JCGO ansågs vara det bästa alternativet för att det var minst komplikationer med att få fram en stabil byggkedja, samt att Java är ett väl utbrett språk bland programvaruutvecklare och ett språk som Raybased vill jobba mer med. En stor fördel är också att JCGO endast behöver översätta de bibliotek i en JVM som används, vilket gör att programmet blir mer minneseffektivt.

Eftersom JCGO endast översätter hela Java-applikationer till C och vi endast vill starta upp ett sådant program en gång, behövs en klasstruktur i Java som kan hantera den meddelandekommunikation som det övriga systemet använder. Anledningen till att bara starta upp det programmet en gång och låta alla javablock ligga i samma program är för att minska den overhead som subtasksJCGO medför vid uppstarten av java-applikationen. Det gör också att sättet som en parenttask fördelar exekveringstid och meddelanden mellan subtasks måste förändras.

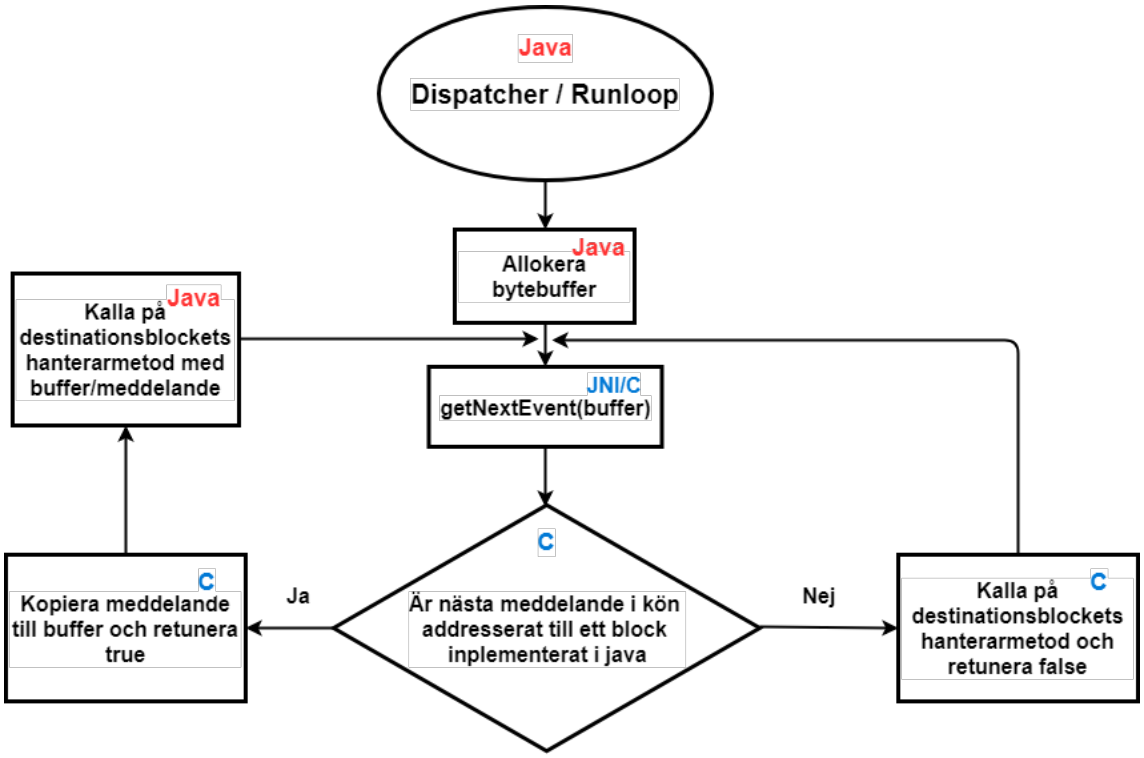
Tillvägagångssättet blir då att flytta logiken för fördelning av exekvering från parenttaskens hanterarmetod till javamiljön. Med hjälp av JNI-metoder kan javamiljön låta block som är implementerade i C exekvera när det blir deras tur. Vid skapandet av en parenttask som har subtasks som är implementerade i både Java och C anges Javas mainmetod som hanterarmetod. Denna förändring påverkar endast parenttasken och dess subtasks. Fördelningen av exekvering för övriga tasks kommer att vara oförändrad och bero på prioritet.



Figur 4.1 - Visar fördelning av subtasks i Java och C för en parenttask

För att köra programmet kontinuerligt används klassen *Dispatcher*. Klassen fördelar meddelanden mellan blocken och anropar hantering av event för det specifika blocket. Detta görs genom en loop. Varje iteration allokerar nytt minne för de meddelanden som ska skickas ut. Ett meddelande är ett event och hämtas via en native-metod *getNextEvent*. Sedan hanteras meddelanden av destinationsblockets hanterarmetod. Beroende på vilket block det är utförs olika event men kan t.ex vara att tända eller släcka en lampa. En viktig byggsten för detta är gränssnittet *OSInterface* vilket innehåller metoderna som blocken kan använda för kommunicera med det underliggande systemet.

För att komma åt underliggande funktionalitet som tillhandahålls av operativsystemet FreeRTOS och funktioner som redan är implementerade används native-metoder. Native-metoder är metoder som är språkoberoende. Det betyder att koden inte kommer att köras via en JVM utan kan köras direkt i språk som t.ex C eller C++. Native-metoder används ofta för att köra systemkommandon eller bibliotek som är skrivna i andra språk och inte finns i det språk som utvecklaren föredrar.



Figur 4.2 - Förenklat flödesdiagram som visar hur födelning av meddelande och exekvering går till för tasks/block implementerade i Java och C.

Genom att använda native-metoder som ett gränssnitt mot funktionalitet som är skriven i C minskar risken för fel. Eftersom att utvecklaren av nya block endast kan skriva kod i Java så finns ingen risk att minnesplatser och pekare manipuleras. Genom att endast skriva native-metoder för den funktionalitet som är nödvändig att komma åt går det att begränsa åtkomligheten till det underliggade systemet.

För att bevisa att konceptet fungerar behövs följande native-metoder

- **long getTime()**
  Metod som kommer att returnera den aktuella tiden direkt från systemklockan. Används inte förtillfället, men är en möjlighet för framtida utveckling.

- **boolean getNextEvent(ByteBuffer buff)**
  Metoden tar en bytebuffer som argument. Om det finns ett meddelande i kön som ska till något av blocken som är implementerade i Java kopieras meddelandet till bufferten *buff* och retunerar true. Finns det meddelanden i kön som ska till block som är implementerade i C fördelas dom till respektive hanterare tills kön är tom. Finns inga meddelanden till javablocken returneras false.

- **void log(String text)**
  Metoden log använder samma metod för loggning som det underliggande systemet.

- **void nativeSendToOutputs(int id, ByteBuffer msg)**

Metoden tar ett meddelande av typen Message som har samma format som RbMessage som de övriga blocken använder. Metoden tar också emot ett id till det block som skickar meddelandet. Metoden lägger därefter ett meddelande till varje block som är listad som output i konfigfilen från ConfigTools i meddelande kö.

4.5 Kompilering

För att optimera och förbättra koden som genererats användes en del flaggor. Dessa flaggor berättar för kompilatorn att utföra eller inte utföra event. Flera av dessa flaggor tar bort beroende som aldrig behövs i detta syfte.

- **NOFILES**: Ge inte åtkomst till det underliggande filsystemet. Ej nödvändigt.
- **NOFATALMSG**: Stänger av JCGOs möjligheter att skriva ut fel om de skulle uppstå. Anledningen till att det stängs av är att man vill använda ett eget system för loggning som man dessutom kan anpassa själv.
- **NATSEP**: Kompilera inte C-filer i JCGOs *native*-bibliotek.
- **NOFP**: Använd inte C-flyttal och dubbeltyper eller flytande aritmetik alls (i detta fall representeras Java-flyttal och dubbeltyper av respektive Java-int och long- typer). Är inte nödvändigt för att projektet och tas därför bort.
- **NOTIME**: Använd inga tidsrelaterade C-funktioner.
- **NOGC**: Använd inte en garbage collector, använd standard calloc istället, så att det tilldelade minnet aldrig återvinns.
- **SEPARATED**: Kompilera alla producerade C-filer separat.
- **NOJNI**: Stöd ej och använd ej Java Native Interface(JNI). JNI kommer ändast att användas genom att kalla på C-funktioner i Java och inte åt andra hållet. Av den enkla anledningen att C inte kan kalla på Java-metoder utan att använda sig av en JVM, därav tar vi bort stöd för det.
- **MAINENTRY**: Explicit specificera deklarationen och namnet av main-funktion i C.

4.6 Java Blocksimulator | JUnit

En viktig byggsten vid utveckling av mjukvara är testning. Därav var målet att skapa ett sätt att testa koden, främst för att se om koden uppfyller syftet men också för att förhindra fel och buggar. För att testa programmet på ett enkelt och

kontinuerligt sätt kom tanken upp om en simulator som ska imitera pucken. Simulatorn ska byggas på samma blocklogik som puckens tilltänkta mjukvara och delar ut olika event till systemet. Tillsammans med simulatorn är målet att använda testningsramverket *JUnit* som på så sätt ska kalla på simulatorn och jämföra resultatet som kommer tillbaka. För ytterligare information under testningen ska ett loggningsblock, ProbeBlock, kopplas mellan två block och vid varje event logga pågående event.

4.7 Testning och verifiering av mjukvara

För testning av systemets funktionalitet har raybased en byggkedja som genererar en binär som är körbar i en linuxmiljö. Det gör det möjligt att testa ny kod utan att ladda in den på en puck vilket är tidskrävande och svårt att felsöka. Med hjälp av *GNU Project Debugger, GDB* har vi stegat igenom övergångarna mellan C och javakoden och hantering av meddelande för att undersöka om systemet beter sig som förväntat. Det har också utförts ett praktiskt test på en puck för att verifiera att översatt kod från JCGO är körbar och att binären är av rimlig storlek.

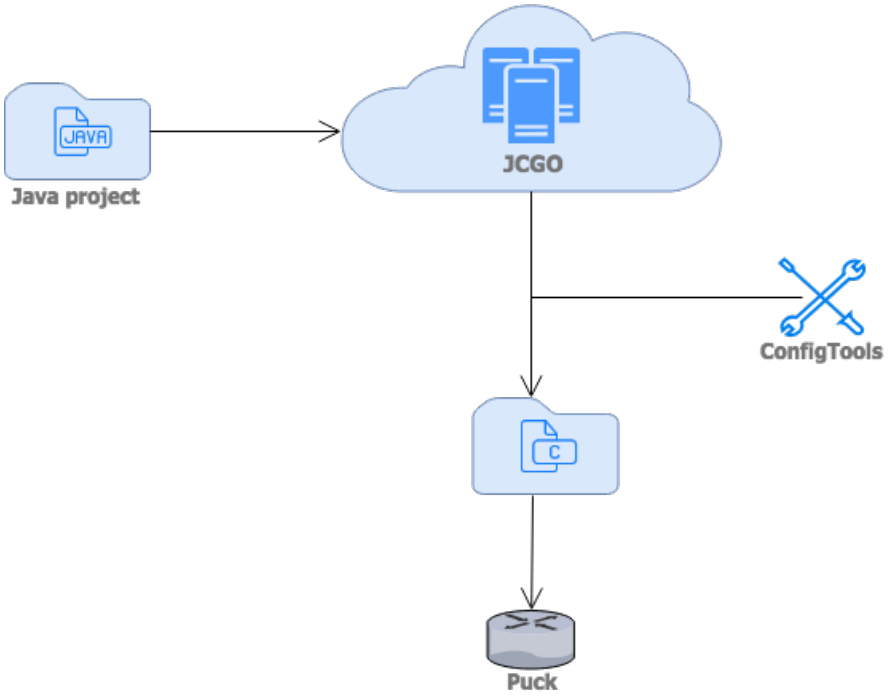
5

Övergripande systemstruktur

Detta kapitel ger en övergripande bild över hur det slutliga systemet är uppbyggt samt en mycket kort beskrivning av dess separata delar.

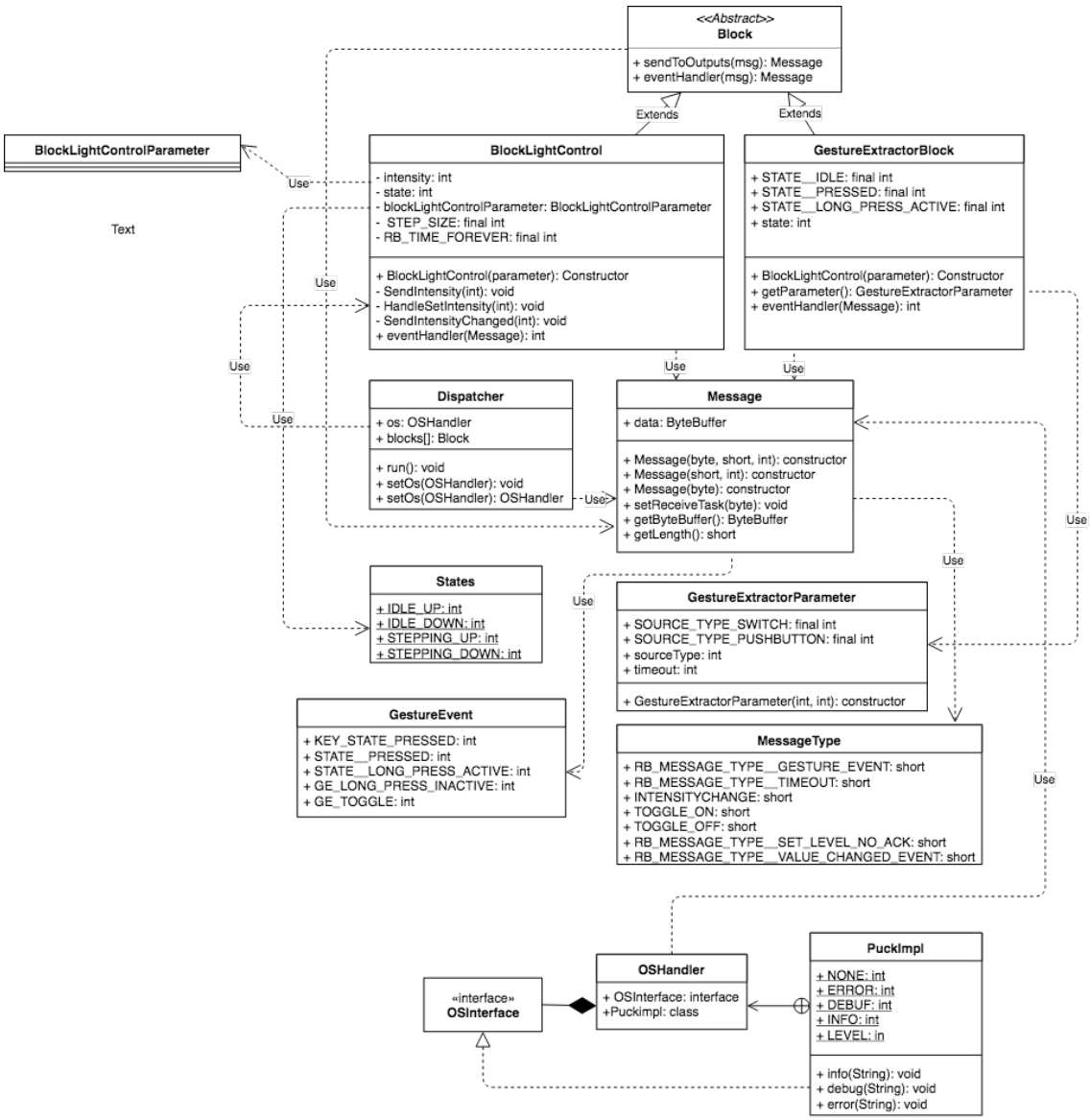
5.1 Systemstruktur

Som grundläggande form kommer systemet att bestå av tre olika fristående delar; koden skriven utifrån, t.ex från kund eller av en utvecklare på Raybased, kompilering och översättning i molnet och plattformen *ConfigTools*.Målet är att JCGO ska vara en del av en service som Raybased tillhandahåller för kompilering av javakod till systemet. Anledningen är att Raybased ska ha full kontroll över den C-kod som har genererats, så att den inte manipuleras efter översättning. Flödet kommer att fungera så att hela Java-applikationen skickas upp till molnsidan för att översättas och kompileras. När detta är utfört utvecklas en lösning i ConfigTools för den specifika byggnaden. ConfigTools genererar en konfigurationsfil skriven i C som behövs för att puckarna ska fungera tillsammans. Denna konfigurationsfil kombineras sedan med den översatta koden. Om det går vägen skapas binärar som sedan går att köra på puckarna. Se figur 5.1.



Figur 5.1 - Övergripande bild av byggkedja

5.2 Blockstruktur - UML

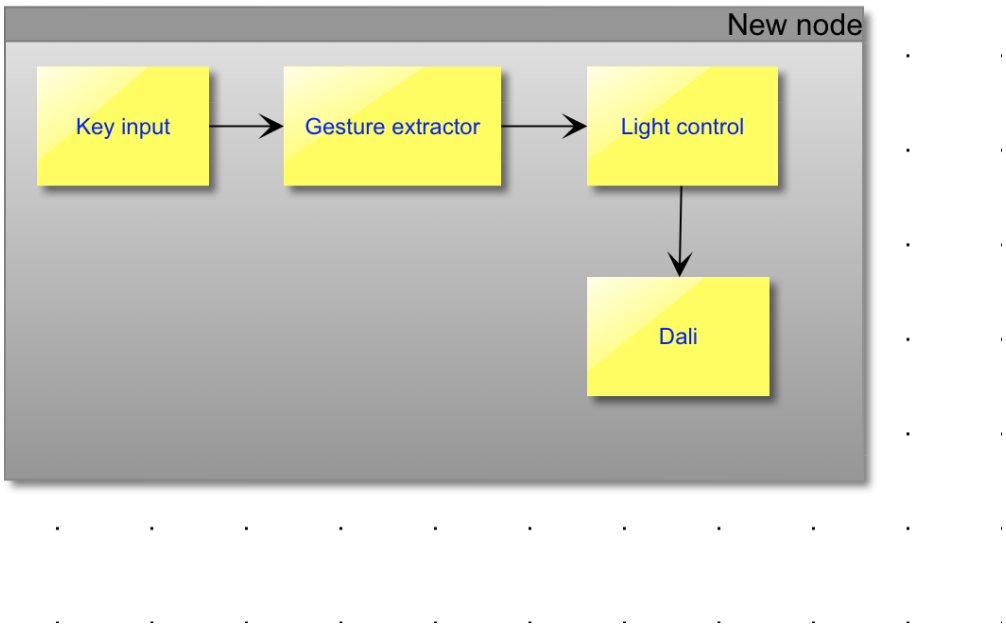


Figur 5.2 - UML-diagram för översatt java-application innehållande blocken GestureExtractor och LightController.

Raybased använder sig av en struktur baserat på moduler eller *block*. Genom att använda block slipper man onödigt stora beroenden och kan istället göra det mer generiskt. Målet är att kunna byta ut ett block till ett annat godtyckligt utan att behöva konfigurera beroenden. T.ex finns möjligheten att byta ut *BlockLightControl* mot något annat block eftersom den ärver från block-klassen.

5.3 ConfigTools

ConfigTools är en plattform utvecklad av Raybased. Dess syfte är att enkelt kunna koppla ihop beroenden och generera kod utifrån de beroenden som finns. Det ger en även möjligheten att få ett överseende över hela produkten och alla dess block.



Figur 5.3 - Övergripande bild på hur ConfigTools ser ut.

Som figur 4.3 beskriver så finns det olika rutor, dessa rutor representerar olika block och mellan dessa block behövs beroende för att säkerställa att rätt block kopplas ihop. Det finns dock redan en viss säkerhet implementerat vilket inte tillåter att vilka block som helst kopplas ihop utan bara kompatibla. T.ex hade det inte varit tillåtet att koppla ihop *Key input* direkt med *Dali*.

När ett projekt har skapats i ConfigTools finns möjligheten att generera en konfigurationsfil. Denna fil innehåller all information som puckarna behöver för att fungera tillsammans för ett specifikt projekt.

6
Diskussion och Resultat

Uppgiften som åtogs var tydlig och specificerad utifrån de krav som sattes. Dock ville inte Raybased stänga några dörrar och det var upp till undersökningen att ta fram en lämplig lösning till problemet. Då det finns många lösningar var det nödvändigt att specificera vad som skulle räknas som en tillräcklig prototyp. För att undersöka vilka alternativ som skulle vara lämpliga startades projektet med undersökning där man tog fram tre olika strategier. Detta kapitel har som syfte att förklara olika tankar om alternativen och dess för och nackdelar.

6.1 Resultat

Kompileringar av en avskalad version har gjorts mot målplattformen för att verifiera att det fungerar och att storleken på binärerna är rimlig. Ett praktiskt test har även utförts på en puck för att se om det startade och kördes, vilket det gjorde. Dock har inga kompileringar av hela applikationen mot en pucken utförts utan har bara testats direkt mot deras testsystem som simulerar en puck.

6.2 Eftertanke

Möjligheten att skriva javakod direkt mot en JVM hade varit den optimala lösningen på problemet som Raybased har, praktiskt sett. Som nämnts tidigare är Java ett av de största språken och är ett språk som nästan alla programmerare kan. Prestandamässigt är Java och C/C++ mer eller mindre likvärdiga när programmen körs på persondatorer. Problemen börjar först när en JVM ska köras på mindre system och med fler kortlivade processer. I dessa fall tenderar Javas *overhead* att bli så stor att C/C++ är mer fördelaktigt i form av minneshantering och exekveringstid. Vid arbete med processorer som har lägre prestanda som t.ex ARM-arkitekturen brukar prioriteringen ligga på att använda så lite minne som möjligt. Raybased har idag inga minnesproblem men för att kunna föra utveckling vidare i framtiden kräver det att minnet används minimalt. Skulle en JVM användas tas förutsättningar för vidareutveckling bort. Efter samråd och diskussion med handledaren kom vi till ett gemensamt beslut att en JVM inte var en optimal lösning på problemet och att vi istället borde utvärdera de andra alternativen.

Från ett mer teoretiskt perspektiv är Rust ett bättre språk att använda. Rust är som tidigare nämnts ett lågnivåspåk och därför mycket effektivare och bidrar till

högre säkerhet. Dessutom slipper man driva det på en JVM. Rust befinner sig fortfarande i ett tidigt utvecklingsstadie jämfört med andra språk, vilket märks tydligt när det kommer till stöd för arkitekturer tillhörande tier 3[1], dit ARM Cortex M0+ räknas in. För att gå runt problemet testades flera olika open source-versioner som påstods ha stöd för ARM-plattformen. Efter flera försök och få framsteg kom vi till slutsatsen att Rust inte är ett språk moget nog för att tillämpas på denna typ av produkt. Dock har Rust potentialen att vara det men behöver vidareutvecklas för att nå samma standard som andra liknande språk har. Rust kan vara ett möjligt alternativ i framtiden för Raybased.

Det sista alternativet som studerades var JCGO som kontra en JVM och Rust, bidrar med en kombination av Java och C. JCGO tar de praktiska aspekterna från Java och effektivitet i C genom att översätta mellan språken. Ur en grundläggade synpunkt så är JCGO väldigt smidigt och det krävs inte mycket, om alls, konfiguration för att översätta. Däremot uppstod det en hel del problem när JCGO skulle appliceras på Raybaseds befintliga system. Det förväntades att lätt kunna skriva kod och översätta och sedan koppla på systemet. JCGO lovar god och effektiv kod men kräver att utvecklaren förstår hur mjukvaran fungerar för att utföra nödvändiga konfigurationer och optimeringar.[8] Förväntningarna på JCGO var att det skulle vara enkelt att kompilera det översatta javaprogrammet tillsammans med den befintliga C-koden. Det var ett svårt och tidskrävande problem. Det tog därför längre tid än väntat att få fram en stabil byggkedja för hela projektet.

Oavsett problem under flera veckor valdes JCGO över de andra alternativen. Detta berodde främst på att JCGO framstod som det bättre alternativet och uppfyllde alla direkta och indirekta krav som fanns. Dessutom vill Raybased jobba mer med Java. Utöver kraven så kändes JCGO seriöst och allt som tillgick, dokumentationen var tydlig och lätt förståelig. Skaparen, Ivan Maidanski, var även trevlig och hjälpvillig under alla mejlkonversationer. Nackdel med de flesta översättare är att de inte längre underhålls, men eftersom JCGO är baserat på öppen källkod, finns möjligheten att ändra och uppdatera JCGO utefter de behov som uppstår.

Det är värt att påpeka att det finns flera olika andra översättare som eventuellt skulle kunna vara lämpliga för detta endamål. Ett exempel är Toba[17] eller J2C[16]. Toba är en av de äldsta vilket kan ha sina fördelar i form av stabilitet. Dock liksom JCGO, har den inte underhållits på flera år och baserar sig på Java 1.1. Genom att använda Java 1.1, försvinner många funktioner som tas för givet i senare versioner av Java. Utöver detta var dokumentationen bristande och svårförstådd. Vidare har vi J2C vilket är en mycket modernare översättare jämfört med Toba och även den baserad på öppen källkod. J2C erbjuder en väldigt detaljerad dokumentation med exempel och olika användningsområden. Detta är även dess nackdel då dokumentationen måste genomgås ordentligt för att förstås jämfört med JCGO som var läsvänlig och lättare att använda.

[1] Arkitekturer delas in olika kategorier beroende på vilket stöd som finns för den plattformen. [14]

6.3 Målens uppfyllnadsgrad

Den viktiga frågan att ta ställning till här är ifall alla mål relaterade till projektet uppfylldes. I det stora hela betraktas uppgiften som uppfylld enligt de krav som sattes i början. Ett fungerande system eller flöde har tagits fram för utveckling av minnessäkra applikationer. Projektet erbjuder inte bara ökad säkerhet utan även en högre tillgänglighet då Raybased nu har möjligheten att programmera i både Java och C. Några få ändringar i prioriteringarna fick dock göras eftersom småproblem hela tiden uppstod. Dessa problem grundade sig i att vi hade fel prioritering under utvecklingen. Istället för att försöka att förstå hur det fungerar och hänger ihop via ett mindre projekt självstående från Raybaseds system, gjordes tester direkt mot det existerande systemet.

Ett sidomål som uppkommit senare i projektet är att det skulle vara lätt att testa funktionaliteten av Java-applikationen via en simulator och testramverket *JUnit*. Strukturen för simulatorn finns där, men tiden räckte inte till för att fullborda då fokus istället låg på att bevisa att översättningen fungerade för Raybaseds system. Vi anser att huvudmålet är uppfyllt och att övriga mål får ses som en möjlighet för utveckling.

På det stora hela anser vi att fokus har varit på rätt prioritering utifrån alla de förutsättningarna. Men som nämnt tidigare räckte inte tiden till för vidareutveckling vilket hade önskats för projektets skull, men även för Raybaseds skull. Detta beror främst på att det är svårt att sätta sig in i ett redan befintligt system och mycket tid har investerats i att förstå den befintliga C-koden.

6.4 Framtida vidareutveckling

Syftet med ett koncept är att visa vad det finns för möjligheter och eventuella lösningar på befintliga problem. En fortsättning på detta projekt skulle vara att vidareutvecklar och göra klart hela plattformen. I detta arbete bör det ingå att det ska vara användarvänligt och aldrig vara några problem med själva systemet samt att det ska vara en fungerande byggkedja. Smidigast är om systemet fungerar direkt med ConfigTools som en befintlig plattform och på så sätt bidra till färre steg.

En en annan del som skulle kunna vidareutvecklas är funktionaliteten på simulatorn. Att ha en fungerande simulatorn ger Raybased möjligheten att redan innan översättningen se eventuella problem och återgälda de direkt. De hade sparat både tid och pengar då risken att problem uppstår efter installation finns.

6.5 Etik och miljöaspekter

Raybaseds system används i olika typer av fastigheter för styrning och bevakning av allt ifrån lampor till ventilation. Av etiska skäl är det viktigt att ett sådant system håller en hög standard och en hög säkerhet. Det är viktigt att den mjukvaran som

används är korrekt och att det inte finns några säkerhetsbrister som gör att systemet går att manipulera eller avlyssna. Med det nya sättet att utveckla mjukvara som har beskrivits i rapporten, minskar risken för denna typen av problem. Detta projektet har också en miljömässig vinst eftersom fler kan utveckla mjukvara till Raybased system vilket gör att fler optimeringar i fastigheter görs som bidrar till minskad energianvändning.

Referenser

[1] Jorge Aparicio. xargo. https://github.com/japaric/xargo/blob/master/README.md, [Accessed 25 May 2018].

[2] Alex Crichton. Rust Once, Run Everywhere, April 2015. https://blog.rust-lang.org/2015/04/24/Rust-Once-Run-Everywhere.html, [Accessed 25 May 2018].

[3] Niklas Adolfsson Fredrik Nilsson. A Rust-based Runtime for the Internet of Things. http://publications.lib.chalmers.se/records/fulltext/250074/250074.pdf, [Accessed 25 May 2018].

[4] Freertos. FREE professionally developed and robust real time operating system for small embedded systems development. https://www.freertos.org/RTOS.html, [Accessed 25 May 2018].

[5] It-ordlistan. Kompilator. https://it-ord.idg.se/ord/kompilator/,[Accessed 25 May 2018].

[6] Ulrich Neumann J.P.Lewis. Java versus C++ benchmarks. http://www.scribblethink.org/Computer/javaCbenchmark.html,[Accessed 25 May 2018].

[7] JUnit. JUnit - About. https://junit.org/junit4/, [Accessed 25 May 2018].

[8] Ivan Maidanski. ivmai/JCGO. https://www.github.com/ivmai/JCGO/blob/master/README, [Accessed 7 February 2018].

[9] Oracle. Java Garbage Collection Basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html,[Accessed 25 May 2018].

[10] Oracle. Java Virtual Machine Technology. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/,[Accessed 25 May 2018].

[11] Ben Putano. Most Popular and Influential Programming Languages of 2018. https://stackify.com/popular-programming-languages-2018/,[Accessed 25 May 2018].

[12] rust lang. Foreign Function Interface - The Rust Programming Language. https://doc.rust-lang.org/nomicon/ffi.html,[Accessed 25 May 2018].

[13] rust lang. Meet Safe and Unsafe - The Rustonomicon. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html, [Accessed 25 May 2018].

[14] rust lang. Rust Platform Support - Rust Forge. https://forge.rust-lang.org/platform-support.html, [Accessed 25 May 2018].

[15] rust lang. What is Ownership? - The Rust Programming Language. https://doc.rust-lang.org/nightly/book/second-edition/ch04-01-what-is-ownership.html, [Accessed 25 May 2018].

[16] Hartmut Schorrig. Java2C - Requirements and features. http://www.vishia.org/Java2C/html/features.html,[Accessed 25 May 2018].

[17] Gregg Townsend Peter Bigot Patrick Bridges Tim Newsham Todd Proebsting, John Hartman and Scott Watterson. Toba: A Java-to-C Translator. https://www2.cs.arizona.edu/projects/sumatra/toba/doc/,[Accessed 25 May 2018].

[18] Hokwon Kim Seok Joong Hwang Youngsun Han, Shinyoung Kim and Seon Wook Kim. Code generation and optimization for java-to-c compilers*.