Mjukvaruutveckling av FreeRTOS-baserade inbyggda system

På hårdvara och molnet

Examensarbete inom Data- och Informationsteknik

LUDVIG SVENSSON
MICHAEL DANG

Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2022

Examensarbete 2022

Mjukvaruutveckling av FreeRTOS-baserade inbyggda system

På hårdvara och molnet

LUDVIG SVENSSON
MICHAEL DANG

CHALMERS

Data- och Informationsteknik
Chalmers University of Technology
Gothenburg, Sweden 2022

Mjukvaruutveckling av FreeRTOS-
baserade inbyggda system
På hårdvara och molnet
LUDVIG SVENSSON
MICHAEL DANG

Handledare: Jonas Duregård, Institutionen för data- och informationsteknik
Examinator: Lars Svensson, Institutionen för data- och informationsteknik

Examensarbete 2022
Data- och Informationsteknik
Chalmers University of Technology
SE-412 96 Gothenburg

Typeset in LaTeX
Gothenburg, Sweden 2022

Förord

Den här rapporten är ett examensarbete som utfördes vid institutionen för Data- och informationsteknik på Chalmers Tekniska Högskola i samarbete med företaget Diadrom. Vi vill tacka Henrik Fagrell och Jonas Hellberg från Diadrom som har stöttat oss under arbetet med deras tekniska expertis. Vi vill också tacka vår handledare Jonas Duregård på Chalmers för handledningen under arbetet.

Michael Dang & Ludvig Svensson, Göteborg, juni 2022

Sammanfattning

Mjukvaruutveckling mot inbyggda system har historiskt varit präglad av begränsad tillgång till fysisk hårdvara. Som ett resultat kan utvecklingen och testerna som görs mot fysisk hårdvara orsaka konflikter och tidskrävande problem mellan teammedlemmar. Diadrom strävar efter att effektivisera utvecklings- och distributionsprocessen genom att utforska möjligheterna att simulera inbyggda system i molnet, så att teammedlemmarna inte begränsas av den fysiska hårdvaran.

Syftet med examensarbetet är att undersöka möjligheterna att simulera hårdvara baserad på FreeRTOS applikationer i molnet. Dessutom: hur kan mjukvaruutveckling av en inbyggda applikationer utformas för att underlätta virtualisering i molnet men samtidigt fungera mot den fysiska hårdvaran?

Undersökningen började med att definiera problemet och sedan gå vidare till att utforma ett lösningskoncept. Innan man kom fram till en lösning behövdes en stor mängd efterforskning och testning göras. Huvudfokusområdet låg kring Amazon AWS och ARM AVH. Efter avslutad efterforskning och testning kunde en prototyp implementeras.

Den praktiska bidraget var ett proof of concept som kördes på hårdvara och AWS. Samtidigt är det teoretiska bidraget ett tillägg till kunskapen om simulering av inbyggd mjukvara som kommer att vara användbar under den kommande framtiden. Sammanfattningsvis har resultaten visat indikationer på att det är möjligt att skapa en simulerad miljö för utveckling av inbyggd applikationsprogramvara som kan representera en fysisk miljö inom en snar framtid

Abstract

Software development on embedded devices is historically constrained by scarce access to target hardware. As a result, the development and testing done on real hardware could cause conflicts and time-consuming problems between team members. Diadrom aims to speed up the development and deployment process by exploring the possibilities to simulate embedded devices in the cloud, so that the team members are not limited by the hardware device.

The purpose of this project is to examine the possibilities of simulating hardware based on FreeRTOS embedded applications in the cloud. Additionally, how can software development of an embedded application be designed to facilitate virtualization in the cloud yet work on the target hardware device?

The examination started off by defining the problem and then moving on to design a solution/concept. Before coming up with a solution, a considerable amount of research and testing had to be done. The main environment of investigation were around Amazon AWS and ARM AVH. Upon finishing the research and testing, a prototype could be implemented.

The practical contribution effort was a proof of concept running on hardware and AWS meanwhile the theoretical contribution is an addition to the knowledge of simulation of embedded software that will be useful in decades to come. In conclusion, the results have shown indications that it is possible to create a simulated environment for embedded application software development that could represent a physical environment in the near future.

Keywords: embedded devices, target hardware, simulating hardware, FreeRTOS, AWS, AVH.

Akronymer

Nedan är en lista på akronymer som använts inom rapporten:

API	Applikationsprogrammeringsgränssnitt
AVH	Arm Virtual Hardware
AWS	Amazon Web Services
BSD	Berkley sockets
GPIO	General-purpose input/output
I/O	Input/Output
SSH	Secure Shell
UART	Universal Asynchronous Receiver/Transmitter
USART	Universal synchronous and asynchronous receiver-transmitter
VHT	Virtual Hardware Target
CI	Continuous integration
CD	Continuous delivery
TCP	Transmission Control Protocol
MQTT	MQ Telemetry Transport
HTTP	Hypertext Transfer Protocol
loRaWAN	Long Range Wide Area Network

Innehåll

Figurer

1
Inledning

1.1 Bakgrund

Diadrom utvecklar mjukvarulösningar för olika sorter av inbyggda system där man vanligtvis enbart har en eller två uppsättningar av hårdvaran. Därmed kan det uppstå en åtkomstkonflikt när flera medarbetare vill använda samma hårdvara. Därför vill företaget undersöka potentialen att simulera delar av hårdvaran i molnet för att underlätta utvecklingen av projekt.

1.2 Syfte

Syftet med uppdraget är att undersöka ifall det är möjligt att simulera hårdvara baserad på FreeRTOS i molnet samt hur låg fördröjning man kan uppnå mot tidskritiska system. Målet med uppdraget är att få fram en virtuell kopia av hårdvaran, reda ut vad för system som kan implementeras samt få fram mätbara värden på den fördröjning som sker och se om den är acceptabel mot tidskritiska system.

1.3 Avgränsningar

Projektet kommer inte att fokusera på att utveckla fysisk hårdvara eller att optimera implementering mot molnet utan snarare se till hur stor del man kan simulera en mikroprocessor.

Simuleringen av hårdvaran kommer att ske genom AWS där det finns stöd för API med FreeRTOS samt för hårdvaran som kommer att användas under projektet.

Eftersom fokus för detta uppdrag är att undersöka hur väl man kan simulera hårdvara i molnet och inte utveckling av mjukvara kommer projektet att vara begränsat till lösningar som redan existerar på marknaden.

1.4 Precisering av frågeställningen

Hur kan ett inbyggt system simuleras i en cloud-miljö med syfte att uppnå utvecklingseffektivitet och minska behovet av fysiska instanser av hårdvara?

Man kan förutsätta att den simulerade versionen kommer bete sig annorlunda än den fysiska. Vad gränserna går för vad som kan betraktas som tillräckligt lika är troligen beroende på den individuella produktens krav. Vi kommer att undersöka skillnader i aspekter såsom i process scheduling, exekveringshastighet, latency och minnesanvändning.

2
Teknisk bakgrund

I detta kapitel beskrivs de nödvändiga begrepp och teknologier som har använts under projektet.

2.1 ARM Virtual Hardware (AVH)

ARM Virtual Hardware är en del av ARMs modelleringsteknologi som tillhandhåller modeller av Arm-baserade processorer, system samt tredjeparts-hårdvara för bland annat applikationsutvecklare. Målet med teknologin är att kunna utveckla och testa mjukvara innan man har fått tillgång till den riktiga hårdvaran [1]. Förutom simulering av hårdvara finns integrering utav CI/CD omgivningar för att testa och verifiera kod.

ARM Virtual Hardware Developer Resource (AVHDR) som är en del av AVH ger utvecklaren tillgång till drivrutiner för gränssnitt som gör att man kan koppla den simulerade processorn mot virtuell utrustning med hjälp av python script. Användningar för dessa funktioner kan kopplas allt från ljudprocessering, maskininlärning till IoT applikationer [2].

2.2 FRDM K64F

FRDM K64F är ett utvecklingskort för Kinetis® platformen utvecklad av NXP semiconductors. Kortet drivs av en mikrokontroller MK64FN1M0VLL12 vilket är baserad på en arkitektur från en ARM Cortex-M4 mikrokontroller [3].

Formfaktorn för i/o portarna på kortet är kompatibel med formfaktorn av en Arduino UNO Rev3 samtidigt som det finns en ethernet port samt två stycken mikro-USB portar där det går att driva kortet genom 5V [3].

För att debugga på kortet finns det en inbyggd adapter som är kopplad till en av mikro-USB portarna. Debuggern heter OpenSDAv2 och den medför att man kan debugga kortet genom datorn, flasha kortet samt hålla en seriell kommunikation mellan dator och kortet [4].

2.3 ARM Cortex-M4

ARM Cortex-M4 är en processor som är designad av ARM och släpptes året 2010 och använder sig av en 32-bit arkitektur. Den används inom mikrokontrollers för allt från system som kräver låg energiförbrukning och låg minneskapacitet till mer krävande system [5]. Processorn är konstruerad för att vara energisnål och har därmed ett antal vilofunktioner för att minska energikonsumtionen [6].

2.4 FreeRTOS

FreeRTOS är ett realtidsoperativssystem som är designat för små mikroprocessorer. Det distribueras under en gratis licens distribuerad av MIT där själva kärnan och ett antal bibliotek inkluderas. Exempel på funktioner som finns att ta del av är TCP, MQTT, HTTP och loRaWAN vilket är olika typer av nätverksprotokoll. Funktioner som finns som standard inom FreeRTOS kärnan är threads och funktioner för att få ut mätvärden som tillexempel ticks, vilket är en intern konfigurerbar klocka som FreeRTOS använder sig av. [7].

3
Metod

3.1 Definiera problemet

Mjukvaruutveckling för inbyggda system sker vanligen med hårdvara som fysiskt finns tillgänglig. Behovet av hårdvaran gör att utvecklingen kan sinkas vid tillfällen där man inte har tillgång till denna. Vid tillfällen där något av de kort eller korten som man har tillgång till under utvecklingen skulle skadas skapas det en flaskhals. Det kan medföra lång väntetid tills man kan fortsätta att utveckla sin kod med hårdvaran.

Liknande problem uppstår när man arbetar med prototyper av inbyggda system, då är det vanligt att man enbart producerar ett lågt antal prototyper då det uppstår en stor kostnad att producera i små serier. Det medför att om det är ett stort team som jobbar med att utveckla mot prototypen så kommer inte alltid de som vill testa sina funktioner ha tillgång till prototypen.

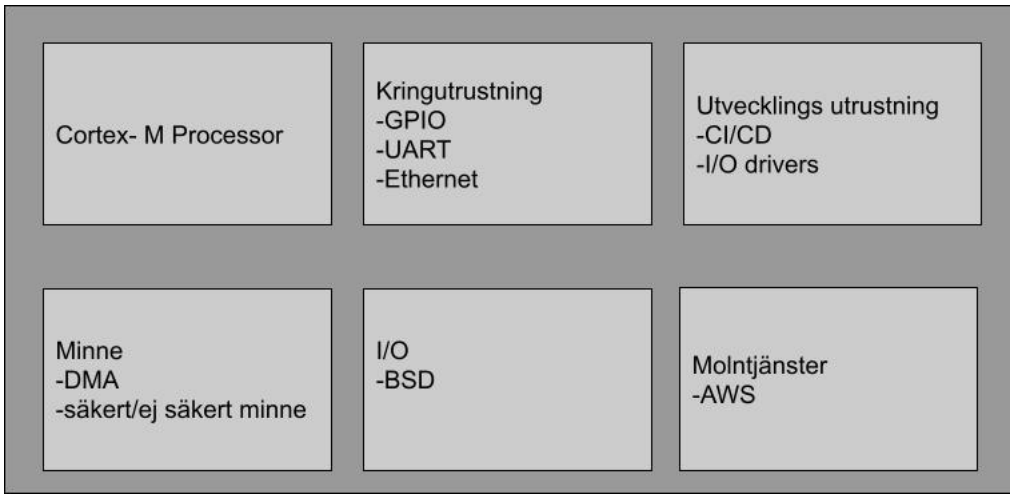
3.2 Identifiering av lösning

För att minska behovet av specifik fysisk hårdvara, vore det önskvärt med något annat alternativ som fungerar liknande eller bättre att utveckla mot och som inte drar ner på utvecklingstiden.

Ett alternativ till att ha tillgång till fysisk hårdvara kan vara att simulera hårdvaran. Detta kan ske på olika nivåer. Olika nivåer på implementation krävs för olika typer av utveckling mot hårdvaran. Det kan vara från att testa hur exekveringshastigheten är på mikroprocessorn till att implementera kontakt med I/O.

Vid implementering av en simulator för en mikroprocessor är det viktigt att man inte enbart får veta att koden kan köras på simulatorn utan att man även får fram inställningar som gör att man sedan direkt kan köra samma kod på den fysiska hårdvaran utan att göra några ändringar. Därför kommer en simulator som heter ARM VHT att användas för att undersöka till vilken detaljnivå som man kan implementera en mikroprocessor samt hur prestandan skiljer sig mot en fysisk motsvarighet.

Enligt Arm VHT är det möjligt att simulera alla deras Cortex-M processorer vilket inkluderar simulering där man använder sig av "riktigt" simulerat minne, I/O portar som BSD sockets samt kringutrustning som GPIO [2].



Figur 3.1: En översikt över några funktioner som inkluderas i VHT

Som man ser i Figur 3.1 ska det existera lösningar för att testa och utveckla kod mot simulerade portar och kommunicera mellan det simulerade kortet och datorn som simuleringen körs på.

Förutom simulering av mikroprocessorn körs koden inom ett lättviktigt realtidsoperativsystem som är designat för att köras på mindre system som mikrokontroller. Operativsystemet som implementeras baseras på FreeRTOS där det finns funktioner för att köra program i trådar och få ut data för att läsa av hur lång tid en funktion eller program har tagit [8]. Så för att få ut data för skillnader och likheter mellan kort och simulatorn kommer all kod som testas köras med FreeRTOS.

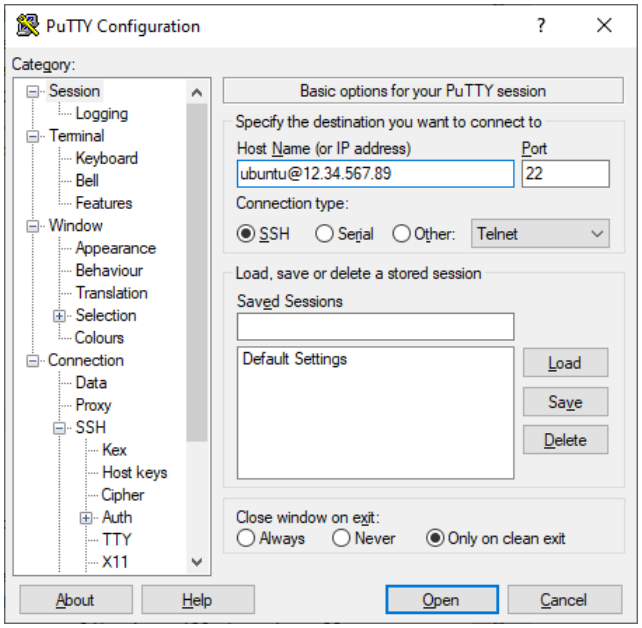
3.3 Implementation av prototyp

Innan man börjar med implementationen så behöver man ta reda på vad för processorarkitektur hårdvaran använder sig av. För detta arbete användes utvecklingskortet FRDM-K64F som är baserat på en ARM Cortex-M4 processor. ARM har en produkt som kallas för Arm Virtual Hardware (AVH) där de tillhandahålla simulerade ARM-processorer, även kallade ARM Virtual Hardware Targets [2].

För att utnyttja dessa simulerade processorer i molnet krävs det att man använder Amazons tjänst AWS Marketplace då de är de enda som stödjer denna ARM-produkt för tillfället. Genom AWS Marketplace startar man en instans som innehåller produkten AVH, vilket är en virtuell server som ligger i AWS Cloud.

Det finns ett fåtal olika sätt att ansluta sig till servern beroende på vilken typ av operativsystem man använder. För detta projekt används kommunikationsprotokollet Secure Shell Protocol (SSH) med hjälp av programmet PuTTY för att på ett säkert sätt ansluta sig till den virtuella servern. Innan man startar en instans krävs

det att man skapar ett så kallat nyckelpar i AWS. Nyckelparet består av två delar, en offentlig nyckel och en privat nyckel. Dessa två nycklar används tillsammans som ens egna säkerhetsuppgifter för att visa instansen att man har rättigheter till att ansluta. Den offentliga nyckeln sparas i själva instansen medan man själv får lägga undan sin privata nyckel, då det är denna man använder sig av för att ansluta sig genom SSH med hjälp av PuTTY och instansens IP-adress.



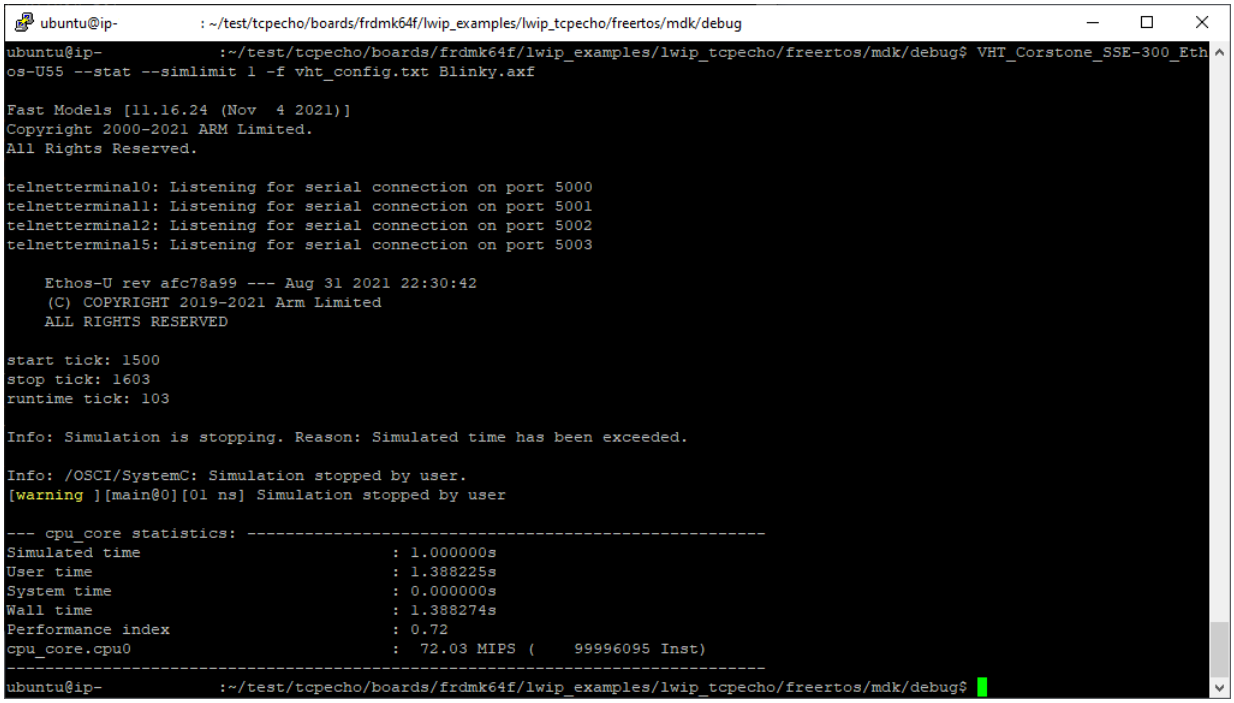
Figur 3.2: Överblick på programmet PuTTY

I Figur 3.2 kan man se hur programmet PuTTY användes för att ansluta till den virtuella servern. Under kategorin SSH på vänstra sidan kan man finna underkategorin Auth vilket ger en alternativet att importera den privata nyckeln för att autentisera.

Operativsystemet Ubuntu vilket är baserat på Linuxkärnan körs på den virtuella servern. För att köra en applikation mot en simulerad processor så behöver man namnet på den simulerade modellen och applikationen som ska köras. Applikationen väljs med hjälp av kommandoradsalternativet -a följt av namnet på den. Ett exempel på hur man kör en applikation mot en simulerad Cortex-M4 är som följande: `VHT_MPS2_Cortex-M4 -a Blinky.axf`. Som man kan se är filen av typen axf vilket krävs för att den simulerade modellen skall kunna köra applikationen. Denna filtyp kan man få ut från Arm Keil MDK vilket är ett program för att skapa applikationer. Kompilatorn i programmet skapar en exekverbar fil i formatet ELF som får filtillägget axf.

För att föra över en applikation till den virtuella servern så användes Github, genom att kopiera repository till servern. Med hjälp av Github funktioner som push och pull är det möjligt att snabbt få uppdaterade versioner av applikationerna. Figur 3.3 visar hur det kan se ut efter en körning i simulatorn. Med kommandoradsalternativet `-stat` får man ut diverse statistiker bland annat om hur lång tid körningen

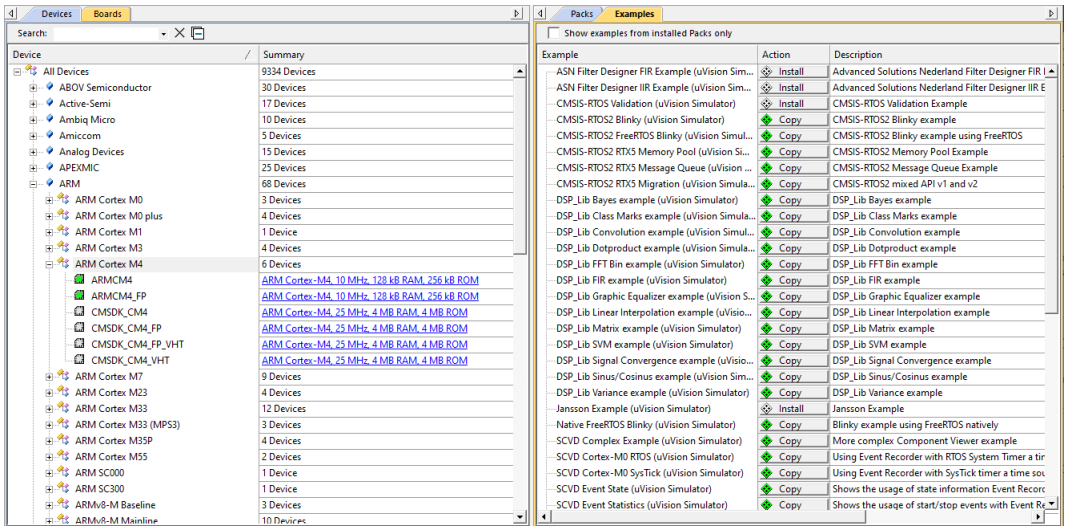
tog för simulatorn och i realtid.



Figur 3.3: Exempel på en körning i simulatorn

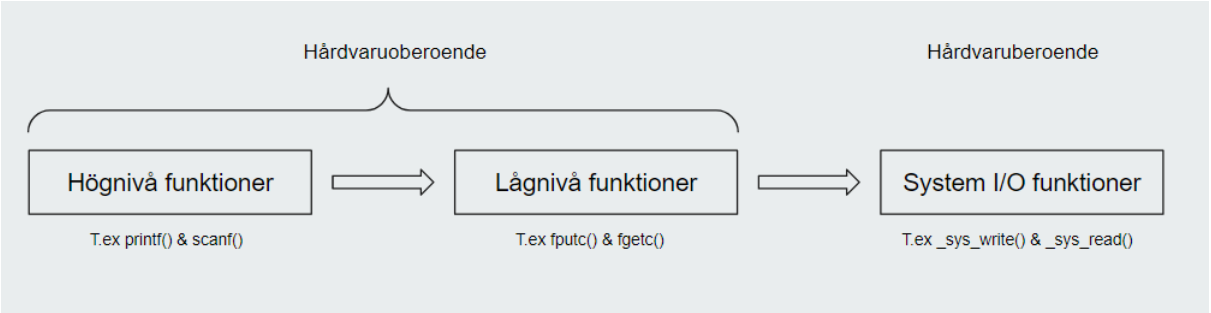
Som man kan se i början av Figur 3.3 så kördes applikationen mot en simulerad modell med namnet `VHT_Corstone_SSE-300_Ethos-U55` och denna modell är baserad på en Cortex-M55 processor. Anledningen till att testerna i simulatorn kördes mot en Cortex-M55 processor istället för Cortex-M4 som den fysiska hårdvaran använder sig av var svårigheter med att implementera funktionen för att kunna printa ut resultat samtidigt som FreeRTOS var integrerat.

Det som testades först var att ladda ner ett färdigt exempel som hade FreeRTOS integrerat på Cortex-M4 processorn från Keil MDKs så kallad Pack Installer. I Figur 3.4 kan man se Pack Installer fönstret och dess flera olika mjukvarupaket som erbjuds, såsom bibliotek och drivrutiner som kan användas för att vidareutveckla. Problemet som uppstod med FreeRTOS-exemplet var att man inte kunde utnyttja sig av `printf()` funktionen för att direkt skriva till kommandotolken i simulatorn. Detta är på grund av att output i simulatorn skickas via USART till telnetterminalen. Telnet är en typ av internetprotokoll som tillåter en användare att ansluta sig till en annan dator virtuellt genom en virtuell terminal [9]. Som man kan se i Figur 3.3 så lyssnar simulatorn på telnet portarna 5000 till 5003 vilket man kan använda sig av för att kommunicera med simulatorn.



Figur 3.4: En överblick på Keil MDKs Pack Installer

För att utföra input/output använder oftast programkoden I/O bibliotek funktioner som till exempel `printf()` och `scanf()`. I Figur 3.5 kan man se strukturen för funktionerna i biblioteket. Högnivå funktionerna utför input/output genom att kalla på lågnivå funktionerna som i tur kallar på system I/O funktionerna. Hög- och lågnivåfunktionerna är icke beroende av hårdvara vilket system I/O funktionerna är, därav använder de första två funktionerna sig av system I/O för att kunna interagera med hårdvaran.



Figur 3.5: Strukturen för funktionerna i standard I/O C run-time biblioteket

ARM erbjuder en funktion som heter I/O Retargeting som ger möjligheten att ändra om I/O funktioner i standard C run-time biblioteket [10]. En möjlig lösning till det tidigare problemet är att modifiera outputen så att den blir USART istället för den vanliga STDOUT men detta gick inte för exemplet mot Cortex-M4 processorn. Varför detta inte gick var på grund av att exemplet för just Cortex-M4 processorn inte hade tillgång till alla nödvändiga drivrutiner och API:er. Dessa drivrutiner och API:er krävs för att kunna komma åt konfigurationsfiler för Cortex-M4 processorn för att sedan kunna genomföra retargeting.

För att lösa problemet hittades ett exempel med FreeRTOS integrerat mot en simulerad Cortex-M55 modell som lyckades att printa direkt mot telnet-terminalen

med hjälp av `printf()` funktionen. Detta exempel modifierades för att sedan köra testen som kommer att presenteras i resultatdelen av rapporten. Anledningen till att exemplet behövde ha FreeRTOS integrerat är att testerna utgår från att jämföra och mäta antalet ticks det tar för att köra testerna i både simulatorn och det fysiska kortet. Ticks i detta fallet innebär system tick, vilket är en tidsenhet som timers och fördröjningar är baserade på i realtidsoperativsystem [11].

4
Resultat

Tabellen 4.1 innehåller en sammanfattning på de tester som genomförts på både kortet och simulatorn där den resterande delen av kapitlet ger mer insyn på hur dessa har genomförts.

Testtyp	Resultat av test
Integrering	Fick ut skrivningar men ej yttre kopplingar
Simulering av mikroprocessor	Fungerar motsvarande arkitektur
Testa exekveringshastighet i form av ticks	Fått ut resultat från både kort och simulatorn

Tabell 4.1: Tabell med sammanfattning av test och resultat

4.1 Integrering av hårdvara

När integreringen av hårdvaran mot simuleringen genomförs fokuseras det på två punkter som ligger till grund för att de andra testen ska gå att genomföra.

- Simulering av processor från samma arkitektur
- Simulering av I/O

När man senare jämför hårdvaran mot den simulerade miljön är det viktigt att man har en simulering av arkitekturen som är så lik som möjligt mot den fysiska hårdvaran, för att få ett beteende som kan jämföras över de två plattformarna.

Som man kan se i Figur 3.1 så finns det funktioner för att simulera säkert minne vilket är minne som går att komma åt och osäkert minne som inte går att komma åt, BSD sockets , GPIO samt mera [2]. Det ger möjlighet att kommunicera med den simulerade mikroprocessorn genom den lokala servern som den ligger på.

4.2 Simulering av mikroprocessor

När det fysiska mikroprocessorn skulle simuleras visade det sig att ett antal funktioner som man kan se i Figur 3.1 ej blivit implementerade i Cortex-M4 modellen. Därmed går det inte att få ut information från simulatorn som exempel utskrivningar av data från programmet.

Resultatet av att information ej gick att få ut från den simulerade Cortex-M4 mikroprocessorn simulerades istället en Cortex-M55 mikroprocessor [12]. Med få ändringar på inställningar av vilket kort man ska kompilera för går det att köra samma grundprogram på både det simulerade M55 och fysiska M4 mikropocessorn.

4.3 Exekveringsprestanda i ticks

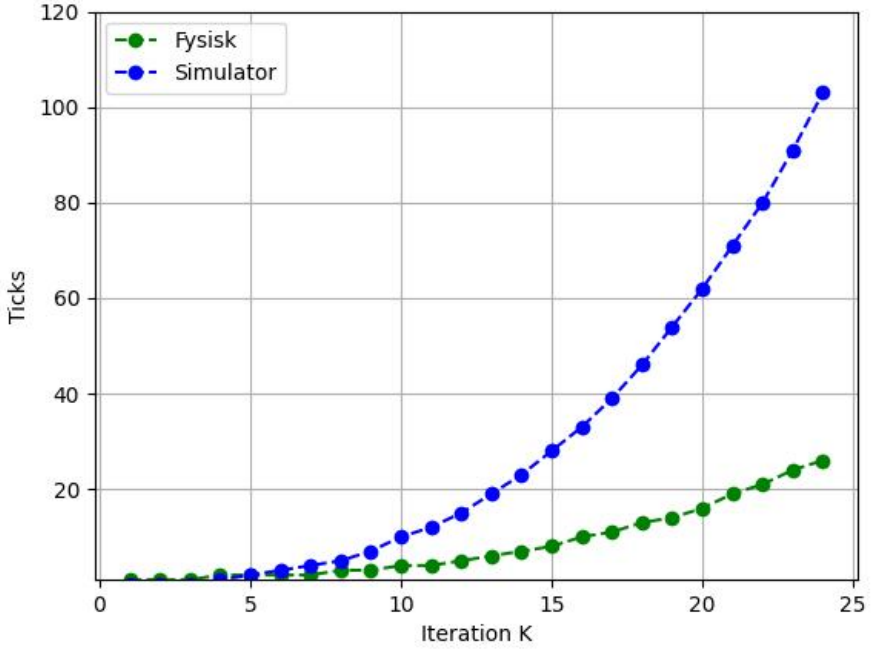
Detta test går ut på att ta reda på hur den simulerade miljön skiljer sig från target när det kommer till hur lång tid det tar att exekvera ett testfall. Resultatet från testet kommer att ge ut hur många ticks som exekveringen tog att genomföra.

Koden för testen som körs i detta kapitel börjar med att skapa en array som innehåller tio slumpmässiga tal från 0 till 99, därefter sorteras denna array från minsta till högsta värdet. Denna biten av kod ligger i en for-loop som ger en möjligheten till att skala upp testen till önskad storlek för att sedan mäta ticksen. I detta test så körs loopen tio gånger där man efter varje loop ökar antalet med tio ytterligare slumpmässiga tal vilket ger en både större och fler arrayer att sortera. Ticksen mäts genom att ta fram det aktuella antalet ticks precis innan arraysen skapas samt precis efter att arraysen har blivit sorterade, sedan tar man skillnaden mellan dessa två värden för att få fram den totala körningen av testet i antalet ticks.

```
1   long starttick = osKernelGetTickCount();
2     printf("start tick: %ld \n", starttick);
3     int sizeArray = 10;
4
5     for(int k = 0; k < 25; k++) {
6       int randomArray[sizeArray];
7       for(int i = 0; i < sizeArray; i++) {
8         randomArray[i] = rand()%100;
9       }
10      for (int i = 0; i < sizeArray; ++i){
11          for (int j = i + 1; j < sizeArray; ++j){
12              if (randomArray[i] > randomArray[j]){
13                int a = randomArray[i];
14                 randomArray[i] = randomArray[j];
15                 randomArray[j] = a;
16              }
17          }
18      }
19
20      sizeArray = sizeArray + 10;
21    }
22
23    long stoptick = osKernelGetTickCount();
24    printf("stop tick: %ld \n", stoptick);
```

```
25
26      long runtick = stoptick - starttick;
27      printf("runtime tick: %ld \n" , runtick);
28      }
29    }
```

Exekvering av koden genomfördes vid 25 iterationer där varje iteration k ökade storleken på arrayen som sorteras. I grafen som kan avläsas i Figur 4.1 representeras den blåa linjen den simulerade mikroprocessorn och den gröna linjen representerar det fysiska kortet. Ett lägre värde på y axeln är något som man vill eftersträva då det representerar en snabbare exekvering.



Figur 4.1: Jämförelse mellan det fysiska kortet och det simulerade kortet i hur många ticks det tagit att genomföra testet

Vid analys av datan under Figur 4.1 syns det att de första iterationerna av testet går ungefär lika snabbt. Där den blåa grafen sedan ökar med en större takt än den gröna grafen. När testet har körts 25 gånger får man ut att vid det största k värdet 25 har simulatorn ticks på 103 och max värde på det fysiska kortet är 26 ticks.

Testet visar att med den prestandan med nuvarande VHT modell är det möjligt att utveckla mot en simulerad processor med minimala eller inga ändringar i koden. Prestandaskillnaderna som uppstår under testet beror med stor sannolikhet på att modellen som användes under testet hade en klockfrekvens som är ca 3.8 ggr långsammare.

5
Diskussion

Som beskrevs i rapporten är simuleringsverktyget som användes under arbetet fortfarande inte helt utvecklat och fortfarande under beta. Detta har medfört att mycket tid gick åt för att undersöka vilka funktioner som fanns tillgängliga för Cortex-M4 processorn under projektet.

När FreeRTOS integrerades mot simulatorn och den fysiska hårdvaran uppkom det en del problem att importera paketet mot både kortet och simulatorn. Även fast det existerade portning mot FreeRTOS i Keil studio så fungerade det inte direkt och krävde en del tid att lösa ut de portningsfel som uppstod.

Som fortsättning på arbetet går det att undersöka hur man skulle gå vidare med att implementera egna funktioner för att få bättre kommunikation ut och in från den simulerade processorn. Exempel på detta hade varit att försöka implementera kommunikation genom UART eller CAN från servern mot simulatorn för att förbättra typen av utveckling som går att köra mot simulatorn och inte kräver att man har tillgång till hårdvaran.

Med fortsatt arbete hade man kunnat implementera GitHub Actions arbetsflödet vilket är ett sätt för att automatisera tester under tiden man mjukvaruutvecklar. Med hjälp av Github Actions hade man kunnat sätta upp arbetsflödet så att varje gång ny kod pushas till ens repository så kör simulatorn testet automatiskt och skickar tillbaka resultat. På detta vis behöver man inte manuellt köra varje kommando som krävs för att genomföra testet varje gång man har uppdaterat koden.

6

Slutsats

6.1 Simulatorn och target

Vid jämförelse av prestandan mellan den fysiska och simulerade mikroprocessorn visar testet att det fysiska kortet klarar av att köra testet på ett mindre antal ticks. Den stora skillnaden i prestanda som man kan se i Figur 4.1 när iterationerna ökar går till stor del att förklara genom att det fysiska mikroprocessorn har en klocka som är ca 4 ggr så snabb som den simulerade.

Eftersom testet använde sig av en annan modell blev resultatet mer av en riktlinje i hur prestanda kan skilja sig mellan det fysiska mikroprocessorn och den simulerade. Resultatet speglar någorlunda att om man får upp klockfrekvenen på simulatorn så kan man få ut ett likvärdigt resultat när det kommer till utveckling på molnet.

Även fast simulatorn är långsammare räknat i ticks så körs det snabbare i användarens tid. Detta gör att utvecklings-effektiviteten kan öka då man kan köra genom programmet på kortare tid än på det fysiska kortet.

Det kom fram under arbetets gång att funktioner som beskrevs i material för simulatorn inte fungerade som de skulle eller inte var implementerade för alla modeller som fanns att välja mellan. Detta kan ha att göra med att simulatorn fortfarande är relativt ny och fortfarande i beta, vilket gör det förståeligt att alla funktioner inte är implementerade än.

Frågeställningen i början av projektet beskrev att minnesanvänding skulle undersökas. När simulatorn implementerades existerade det underlag på hemsidan för VHT där det fanns en möjlighet att på simulatorn köra en debugger för att få ut information från processorn. Detta fungerade emellertid inte och därmed blev resultatet ett ickeresultat.

Hantering utav latens undersöktes inte när simulatorn testades, eftersom ingen kontakt kunde uppnås med simulatorn medan den körde och därmed inte få ut mätningar på hur processorn skulle reagerat om den fick en insignal.

6.2 Proof of concept

För stunden så är detta ett proof of concept där man kan utveckla mot en simulerad miljö som ska motsvara en fysisk miljö. Med lite mer utveckling anser vi att ett liknande upplägg i framtiden kan bidra till snabbare utveckling av inbyggda system, men att det fortfarande har en bit att gå när det kommer till att implementera I/O och snabbare processormodeller som motsvarar de som man kan få tag på. Det teoretiska bidraget till nuvarande existerande kunskaper om ämnet kommer att vara väl användbara i framtiden.

Litteratur

[1] ARM, *Virtual Hardware – Software Development Without Hardware*, 2022. URL: https://www.arm.com/products/development-tools/simulation/virtual-hardware (hämtad 2022-06-07).

[2] ARM, *Overview: Introduction*, 2022. URL: https://arm-software.github.io/AVH/main/overview/html/index.html (hämtad 2022-06-01).

[3] NXP, *FRDM-K64F Platform/Freedom Development Board/Kinetis MCUs | NXP Semiconductors*. URL: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F (hämtad 2022-04-05).

[4] H. Anthony. (2014). "OpenSDAv2 - NXP Community", URL: https://community.nxp.com/t5/Kinetis-Microcontrollers/OpenSDAv2/ta-p/1121569 (hämtad 2022-06-07).

[5] J. Yiu, *The Definitive Guide to ARM CORTEX-M3 and CORTEX-M4 Processors (Third Edition)*, Third Edition. Oxford: Newnes, 2014. URL: https://www.sciencedirect.com/science/article/pii/B9780124080829000014 (hämtad 2022-06-06).

[6] H. P B, S. R. Anireddy, J. F T och V. R, "Introduction to ARM processors & its types and Overview to Cortex M series with deep explanation of each of the processors in this Family", i *2022 International Conference on Computer Communication and Informatics (ICCCI)*, 2022, s. 1–8. URL: https://ieeexplore.ieee.org/document/9740768/ (hämtad 2022-06-06).

[7] FreeRTOS, *RTOS - Free professionally developed and robust real time operating system for small embedded systems development*. URL: https://www.freertos.org/RTOS.html (hämtad 2022-06-07).

[8] FreeRTOS, *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. URL: https://www.freertos.org/ (hämtad 2022-04-05).

[9] IBM. (2022). "Telnet Protocol - IBM Documentation", URL: https://www.ibm.com/docs/en/aix/7.1?topic=protocols-telnet-protocol (hämtad 2022-06-08).

[10] Keil, *I/O Retargeting: I/O Retargeting*. URL: https://www.keil.com/pack/doc/compiler/RetargetIO/html/index.html (hämtad 2022-06-08).

[11] Keil. (2022). “Systick Timer (SYSTICK)”, URL: https://www.keil.com/pack/doc/cmsis/Core/html/group__SysTick__gr.html (hämtad 2022-06-08).

[12] ARM. (2022). “Cortex-M55 coremark”, URL: https://developer.arm.com/Processors/Cortex-M55 (hämtad 2022-05-30).

A

Kod för att grafiskt visa värderna från bilagorna B-C

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numpy import genfromtxt, maximum
4
5 Target = genfromtxt('Ticks_target.txt', delimiter=",",
    skip_header = 1)
6 Simulator = genfromtxt('Ticks_simulator.txt', delimiter=",",
     skip_header = 1)
7
8 time = list(range(1,25))
9
10 plt.ylim((0,110))
11
12 plt.plot(time , Target , "—bo",label= "Fysisk" , color = '
    green' )
13 plt.plot(time , Simulator , "—bo",label= "Simulator" ,
    color ='blue' )
14 plt.legend()
15 plt.xlabel('Iteration K')
16 plt.ylabel('Ticks')
17 plt.grid(True)
18 plt.show()
```

B

Ticks per iteration på simulatorn

Iteration	Ticks
1	0
2	0
3	0
4	0
5	1
6	2
7	3
8	4
9	5
10	7
11	10
12	12
13	15
14	19
15	23
16	28
17	33
18	39
19	46
20	54
21	62
22	71
23	80
24	91
25	103

C

Ticks per iteration på target

Iteration	Ticks
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	2
9	3
10	3
11	4
12	4
13	5
14	6
15	7
16	8
17	10
18	11
19	13
20	14
21	16
22	19
23	21
24	24
25	26

D

Main program testkod

```
1  /*
2   * Copyright (c) 2013 - 2015, Freescale Semiconductor, Inc.
3   * Copyright 2016-2017 NXP
4   * All rights reserved.
5   *
6   * SPDX-License-Identifier: BSD-3-Clause
7   */
8
9  #include "fsl_device_registers.h"
10 #include "fsl_debug_console.h"
11 #include "pin_mux.h"
12 #include "clock_config.h"
13 #include "board.h"
14 #include "cmsis_os.h"
15 #include "cmsis_os2.h"
16 #include "MK64F12.h"
17 #include <stdio.h>
18 #include "RTE_Components.h"
19 #include CMSIS_device_header
20
21 osThreadId_t tid_phaseA;                    /* Thread id of
       thread: phase_a      */
22
23 osThreadId_t tid_clock;                     /* Thread id of
       thread: clock        */
24
25 struct phases_t {
26   int_fast8_t phaseA;
27 } g_phases;
28
29
30 /*
     ------------------------------------------------------------------------

31  *       Function 'signal_func' called from multiple threads
32  *-----------------------------------------------------------------------
       */
```

```
33 void signal_func (osThreadId_t tid)  {
34   osThreadFlagsSet(tid_clock, 0x0100);       /* set signal to
         clock thread     */
35   osDelay(500);                              /* delay 500ms
                          */
36   osThreadFlagsSet(tid_clock, 0x0100);       /* set signal to
         clock thread     */
37   osDelay(500);                              /* delay 500ms
                          */
38   osThreadFlagsSet(tid, 0x0001);             /* set signal to
         thread 'thread' */
39   osDelay(500);                              /* delay 500ms
                          */
40 }
41
42
43 /*
    ----------------------------------------------------------------------------

44  *       Thread 1 'phaseA': Phase A output
45  *----------------------------------------------------------------------------
     */
46 void phaseA (void *argument) {
47   for (int i = 0 ; i<1 ; i++) {
48     osThreadFlagsWait(0x0001, osFlagsWaitAny ,osWaitForever)
        ;   /* wait for an event flag 0x0001 */
49     g_phases.phaseA = 1;
50     signal_func(tid_phaseA);
                                                /* call common
        signal function    */
51     g_phases.phaseA = 0;
52
53
54     long starttick = osKernelGetTickCount();
55     PRINTF("start tick: %d \n", starttick);
56     int sizeArray = 10;
57
58
59
60       for(int k = 0; k < 10 ; k++) {
61       int randomArray[sizeArray];
62       for(int i = 0; i < sizeArray; i++) {
63         randomArray[i] = rand()%100;
64
65       }
66
```

```
67
68
69        for (int i = 0; i < sizeArray; ++i){
70           for (int j = i + 1; j < sizeArray; ++j){
71              if (randomArray[i] > randomArray[j]){
72                int a = randomArray[i];
73                 randomArray[i] = randomArray[j];
74                 randomArray[j] = a;
75              }
76           }
77        }
78
79
80        sizeArray = sizeArray + 10;
81     }
82
83
84     long stoptick = osKernelGetTickCount();
85     PRINTF("stop tick: %d \n", stoptick);
86
87     long runtick = stoptick - starttick;
88     PRINTF("runtime tick: %d \n" , runtick);
89
90     }
91   }
92
93
94
95
96
97
98
99
100 /*
    ------------------------------------------------------------------------------
101  *       Thread 5 'clock': Signal Clock
102  *------------------------------------------------------------------------------
     */
103 void clock (void *argument) {
104   for (;;) {
105     osThreadFlagsWait(0x0100, osFlagsWaitAny, osWaitForever)
        ;   /* wait for an event flag 0x0100 */
106     osDelay(80);
                                                          /*
        delay 80ms                    */
```

```
107   }
108 }
109
110 /*
      ----------------------------------------------------------------------
111  *       Main: Initialize and start the application
112  *---------------------------------------------------------------------
     */
113 void app_main (void *argument) {
114   tid_phaseA = osThreadNew(phaseA, NULL, NULL);
115
116
117   tid_clock  = osThreadNew(clock,  NULL, NULL);
118
119   osThreadFlagsSet(tid_phaseA, 0x0001); /* set signal to
        phaseA thread    */
120
121   osDelay(osWaitForever);
122   while(1);
123 }
124
125 /*
      ----------------------------------------------------------------------
126  *       Main: Initialize and start the RTOS2 Kernel
127  *---------------------------------------------------------------------
     */
128 int main (void) {
129
130   //char ch;
131
132     /* Init board hardware. */
133     osKernelInitialize();
134     BOARD_InitBootPins();
135     BOARD_InitBootClocks();
136     BOARD_InitDebugConsole();
137     SystemCoreClockUpdate();
138     PRINTF("new run \n");
139
140
141   // System Initialization
142
143
144                     // Initialize CMSIS-RTOS
```

```
145     osThreadNew(app_main, NULL, NULL);     // Create
            application main thread
146     if (osKernelGetState() == osKernelReady) {
147       osKernelStart();                     // Start thread
              execution
148     }
149
150     while(1);
151   }
```

Data- och Informationsteknik
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS