EXAMENSARBETE INOM ELEKTROTEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2021*

Emulering av c-applikationer för ett inbyggt system i Linuxmiljö

Emulation of c applications for an embedded system in Linux

MARIKA LOGGE

KTH
SKOLAN FÖR KEMI, BIOTEKNOLOGI OCH HÄLSA

Emulering av c-applikationer för ett inbyggt system i Linuxmiljö

Emulation of c applications for an embedded system in Linux

MARIKA LOGGE

Examensarbete inom
Elektroteknik
Grundnivå, 15 hp
Handledare på KTH: Linus Remahl
Examinator: Elias Said
TRITA-CBH-GRU-2021:052

KTH
Skolan för kemi, bioteknologi och hälsa
141 52 Huddinge, Sverige

Sammanfattning

I det här arbetet har en emulator till DeLavals inbyggda system IOM 200 utvecklats i en Linuxmiljö. Konceptet har varit att implementera en emulator i DeLavals testprocess för mjukvaran i ett inbyggt system. Syftet med emulatorimplementationen var att underlätta utvecklingen av mjukvaran genom att ta bort beroendet av hårdvaran.

Baserat på studier av olika metoder, tillgängliga verktyg och tidigare arbeten skapades en emulatormodell för IOM 200 och ett koncept för hur den ska implementeras. Arbetet har även skapat en fungerande prototyp som kan exekvera ett mindre kodsegment från IOM 200 och därigenom validerar emulatormodellen.

Emulatormodellen utformades på den redan befintliga FreeRTOS-simulatorn som finns tillgänglig i Linux. Anledningen är att FreeRTOS används i IOM 200, den är gratis att använda och den möter emulatorns abstraktionskrav. Utöver FreeRTOS-simulatorn implementerades stubbar och wrapperfunktioner som tillhandahöll gränssnitt som gjorde IOM 200 applikationen exekverbar i emulatorn.

Nyckelord
Emulator, inbyggda system, Linux, FreeRTOS

Abstract

In this work an emulator for DeLaval's embedded system IOM 200 has been developed in a Linux environment. The concept was to implement the emulator in DeLaval's software test process for embedded systems. The purpose of creating an emulator was to ease the development of the embedded software by removing the dependency on embedded hardware.

An emulator model and a concept for its implementation was created through the studies of various methods, available tools, and existing works in the emulator field. Based on the model the work created a working prototype that can execute a smaller code segment from the IOM 200 application.

The emulator model was designed on the already existing FreeRTOS simulator that is available for Linux. The motive being that FreeRTOS is the operating system running on IOM 200, it is open source, free to use and it has the perfect level of abstraction for the emulator. Stubs and wrappers were implemented to the emulator in addition to the FreeRTOS simulator. These stubs and wrappers provided the interfaces needed for the IOM 200 application to be executable in the emulator.

Keywords
Emulator, embedded systems, Linux, FreeRTOS

Akronymer

LIN	Local Interconnect Network
DCL	Digital Current Loop
CCM	Control & Communication Module
IOM	Input Output Modul
BM	Button Modul
IO	Input/Output
MPC	Milking Point Controller
ARM	Advanced RISC Machine
PSOC	Programmable System-on-Chip
BLE	Bluetooth Low Energy
GPIO	General purpose Input/Output
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
I2C	Inter-Integrated Circuit
RISC	Reduced Instruction Set
CPU	Central Processing Unit
NVIC	Nested Vector Interrupt Control
ALU	Aritmetisk logisk enhet
JTAG	Joint Test Action Group
WIC	Wakeup Interrupt Controller
PWM	Pulse Width Modulation
RTOS	Real Time Operating System
OS	Operating System
POSIX	Portable Operating System Interface for Computing Environment
IEEE	Institute of Electrical and Electronics Engineers
RTL	Register Transfer Level
DSM	Design Simulation Model
LED	Light Emitting Diode
CRT	C Runtime Library
GUI	Graphic User Interface
VDEES	Virtual Development Environment for Embedded Software
GNU	GNU's Not Unix
SID	Shared Information Data

Förord

Det här examensarbetet har utförts under våren 2021 hos företaget DeLaval. Företaget erbjuder hela eller delvis automatiserade lösningar för allt inom mjölkproduktion och djurhållning. Om läsaren önskar veta mer om DeLaval eller de automatiserade mjölkningstallen som nämns i den här rapporten hänvisas läsaren till DeLavals hemsida (www.delaval.com).

Examensarbetet berör ämnena emulering och inbyggda system där det underlättar om läsaren har övergripande förkunskaper inom området.

Jag vill tacka DeLaval för möjligheten att genomföra det här arbetet. Ett särskilt tack till Dan Zemack för sitt engagemang och vägledning under arbetets fortgång. Jag vill också tacka Pontus Häger och Daniel Brun för ert stöd och resurser till arbetet.

Avslutningsvis vill jag tacka min handledare Linus Remahl på Kungliga tekniska högskolan (KTH) för den akademiska handledningen av arbetet.

Innehållsförteckning

1 Inledning

1.1 Problemformulering

DeLaval är ett globalt och världsledande företag inom mjölkningsindustrin som kan erbjuda flera lösningar för en helt eller delvis automatiserad mjölkproduktion [1]. För att styra och hantera funktioner i en automatiserad verksamhet integreras så kallade inbyggda system. Ett inbyggt system består av både hårdvara och mjukvara som är designat för att uppfylla en eller flera specifika funktioner.

Utvecklingen av mjukvaran för ett inbyggt system kan bli både omständligt och tidskrävande då det finns ett beroende av att testa med hårdvaran. Beroendet tillkommer då programmet bland annat behöver tillgå periferienheter, operativsystem och processorer som sköter exekveringen av koden. Hos DeLaval utvecklas mjukvaran för det inbyggda systemet i en Linuxmiljö för att sedan exekveras på ett testkort för att verifiera kodens funktion. Idag finns möjligheten att utföra enhetstester i utvecklingsmiljön men testandet av mjukvarans beteende måste ske på ett målkort. Med mjukvarans beteende menas programmets funktionalitet, att mjukvaran gör det den ska när den ska göra det.

När mjukvaran ska testas på ett målkort måste hårdvarugränssnittet av koden beaktas. Det leder till att en större mängd kod måste utvecklas före en körning för att det ska fungera korrekt. Komplikationer kommer när ett test genererar ett dåligt resultat och felsökande måste täcka både programmets beteende och dess hantering av hårdvarugränssnittet. Ytterligare komplikationer kan uppstå när hårdvaran kanske inte är tillgänglig eller om den är otymplig. Därför vore implementationen av en emulator fördelaktigt då det kan underlätta testprocessen och förbättrar kvalitén i arbetet. Då en emulator av det inbyggda systemet kan exekvera mjukvaran och möjliggöra testandet av applikationens beteende i Linuxmiljön innan en körning på målkortet.

1.2 Målsättning

Arbetets mål är att utveckla en modell och en prototyp av en emulator för DeLavals inbyggda system IOM 200 i Linux. Emuleringens syfte är att underlätta och effektivisera företagets process för testandet av mjukvarukod till ett inbyggt system. Emulatorns avsikt är att möjliggöra testandet av mjukvarukodens beteende oberoende av den fysiska hårdvaran. Konceptet för emulatorn ska utvecklas genom en väl definierad beskrivning av det inbyggda systemet tillsammans med underlaget från en litteraturstudie. Studien ska beskriva vad Linux är och undersöka tillgängliga simuleringsverktyg som är relevanta för systemet IOM 200. Studien ska även undersöka forskning och andra arbeten inom emulering och simulering vars lösningar är inspirerande eller tillämpningsbara i detta arbete. Vidare ska emulatorns potential till en generalisering analyseras och bedömas.

1.3 Avgränsningar

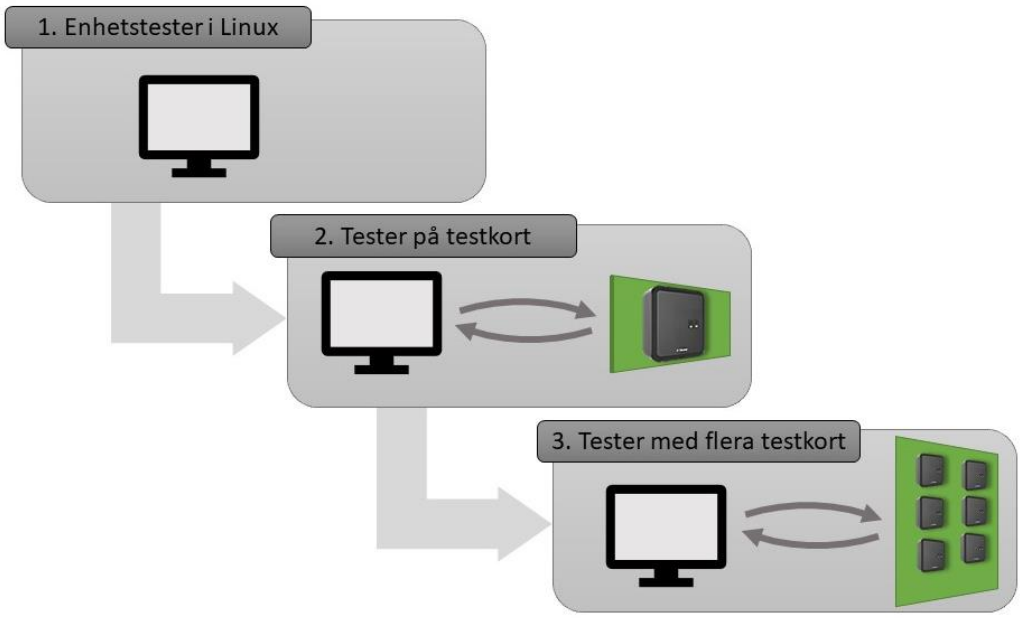
Emulatorn kommer att utformas och anpassas efter applikationen i DeLavals inbyggda system av typen IOM 200. Emulatorprototypen kommer bara att realisera delar av applikationen i systemet medan modellen kommer att beskriva hela lösningen. Emulatorn kommer att testa applikationens beteende utan att ta hänsyn till mjukvarans tidsaspekter eller hårdvarugränssnitt. Verklighetsanpassade simulationer av tidsåtgång eller eventuella tidskritiska aspekter kommer inte att beaktas av arbetet eller realiseras i emulatorn. Funktioner för hårdvaruabstraktion och gränssnitt, så som exempelvis adressering och registerhantering kommer att ersättas med programstubbar. Arbetet kommer inte undersöka eller förverkliga någon avbrottshantering i emulatorn.

2 Teori och bakgrund

Under det här kapitlet presenteras arbetets förstudie och litteraturstudie i fyra avsnitt. I första avsnittet sker en introduktion till DeLavals arbetssätt för att testa koden till ett inbyggt system. I avsnitt 2 redovisas det inbyggda systemet IOM 200 som arbetet kommer att behandla. Där ges först en beskrivning av systemets funktion och syfte för att sedan redovisa systemets hårdvara och mjukvara. I avsnitt 3 beskrivs operativsystemet Linux och Ubuntu som är den Linuxmiljö som användes av arbetet. I avsnitt 4 presenteras existerande lösningar och koncept för emuleringar/simuleringar av inbyggda system på en virtuell plattform.

2.1 DeLavals testprocess för kod till ett inbyggt system

Idag utvecklas mjukvaran till de inbyggda systemen i en Linuxmiljö (Ubuntu 18.04). I Linux används programmet Eclipse som utvecklingsmiljö. Processen för att testa mjukvarukoden kan delas upp i tre delmoment som utförs succesivt. Det sker fortsatta tester efter dessa tre moment men de kommer inte att täckas utav arbetet då dessa tester sker mot riktiga system och gårdar. De tre momenten i testprocessen finns illustrerat i figur 2.1 och beskrivs i nästkommande avsnitt.



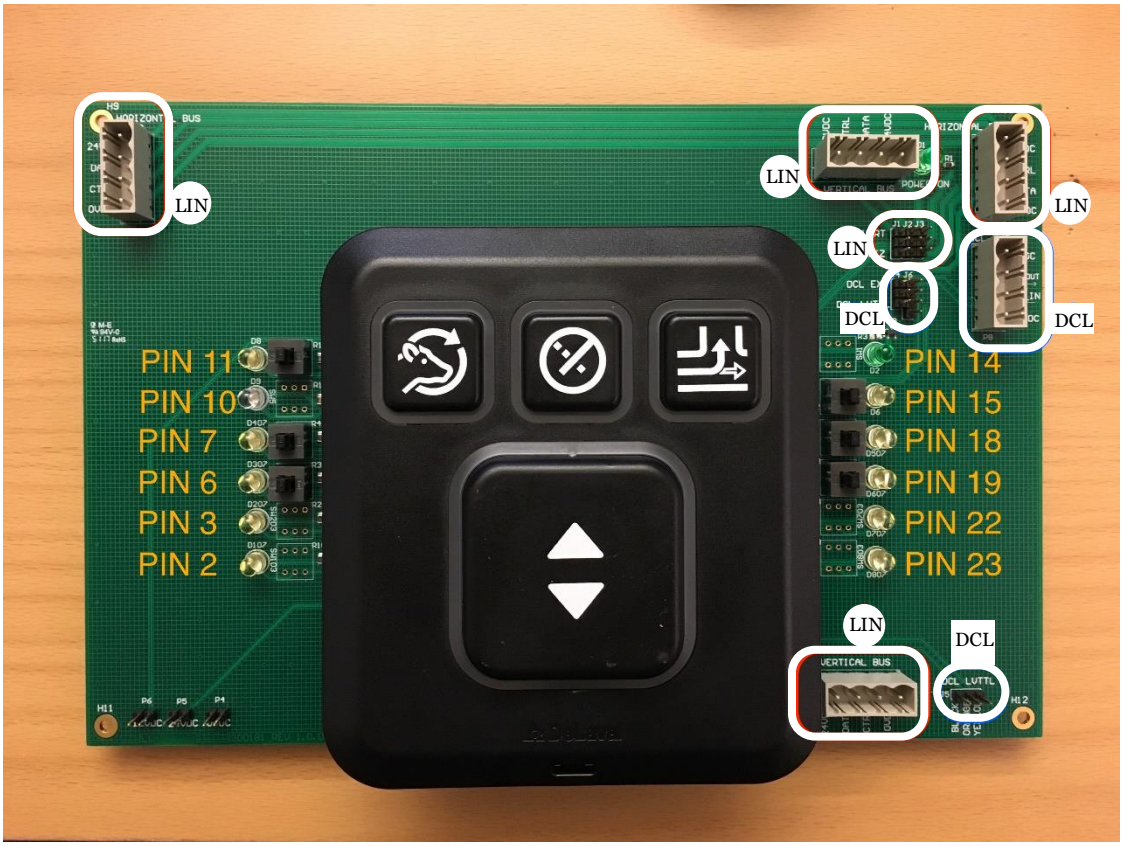
Figur 2.1. Testprocessen hos DeLaval för programvaran till ett inbyggt system.

2.1.1 Enhetstester

De första tester som görs av nyutvecklad kod är så kallade enhetstester. Enhetstesterna ska validera särskilda kodenheters funktionalitet, så som beteendet hos en metod eller funktion. Valideringen görs genom särskilt anpassade testfall mot kodenheten. Testfallen ska vara exekverbara enskilt och oberoende av andra funktioner [2]. Det uppnås bland annat genom att ersätta indata med fasta och simulerade värden, ofta med data som testar eventuella randvillkor.

2.1.2 Testkort

Efter enhetstesterna ska programvarans processer testas. Med processtester menas kontroller av programmets hantering av funktioner, sekvensenlighet och att den löser uppgifterna korrekt. Hos DeLaval testas idag programvarans processer på ett testkort. När programmet exekveras på testkortet sker också en interaktion med hårdvarugränssnittet, vilket innebär att alla delar av programmet som berör hårdvara måste beaktas innan ett test. I Figur 2.2 nedan finns en bild på ett av DeLavals testkort med ett annat inbyggt system av typen BM 213.

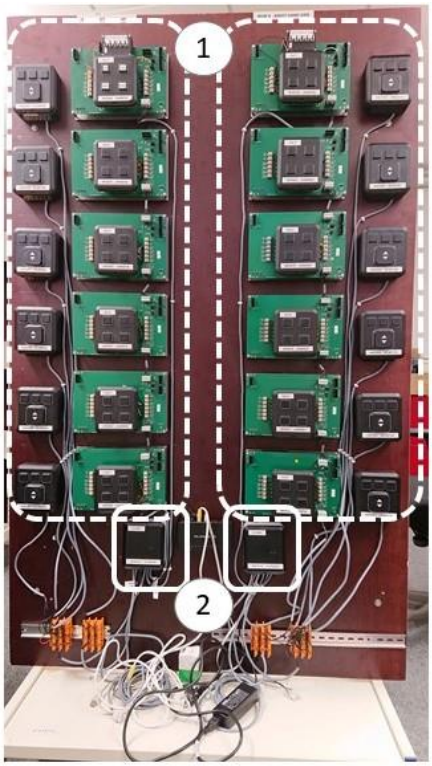


Figur 2.2: DeLaval testkort med BM 213.

Testkortet består av ett inbyggt system (lådan i figur 2.2) som är monterat på ett kretskort (kortet bakom lådan i figur 2.2). Det inbyggda systemet har kommunikation med utomstående enheter av olika slag, kan exempelvis vara en sensor för mjölkflöde eller ett annat system. På testkortet finns det någon typ av substitut för alla gränssnitt med extern kommunikation. För alla utgångspinnar finns en lysdiod (finns vid alla numrerade PIN på testkortet i figur 2.2), på så sätt visualiseras pinnarnas tillstånd vid en testkörning. För de pinnar som agerar ingångar så finns det också en knapp som möjliggör en tillståndförändring hos pinnen. På kortet sitter kontakter för två typer av datakommunikation, där ena är för LIN och den andra är DCL (de är inringade i figur 2.2). LIN är en akronym för Local Interconnect Network och är en enkel databusstyp som använder en tråd för datatransmission [3]. LIN-bussarna används av systemet för att kommunicera med andra enheter i sitt nätverk. DCL är en akronym för Digital Current Loop och är ett gränssnitt som används för att simulera testfall och för att konfigurera och debugga enheten.

2.1.3 Simulering och testväggar

Efter framgångsrika tester av programmet på testkortet så går utvecklingen vidare till något som på DeLaval kallas testväggar. Dessa väggar kan se olika ut och är byggda på så sätt att de ska simulera en eller flera mjölkstationer, ett system eller ett stall. I figur 2.3 nedan finns ett foto av en testvägg som simulerar flertalet mjölkplatser i ett mjölkningsstall. En beskrivning av mjölkningsstall ges i avsnitt 2.2.1 Syfte och funktion.



Figur 2.3. Bild på en testvägg som simulerar flera mjölkplatser i ett mjölkningsstall.

Systemen som är inringade med en streckad vit linje och är numrerat med 1 ska simulera mjölkplatser i ett mjölkningsstall. Den ena raden består av system på testkort för att visualisera statusen på de portar som inte kan anslutas till sina periferienheter. En port kan exempelvis vara ett kommunikationsgränssnitt till en mjölkflödesmätare som inte finns med i simuleringen.

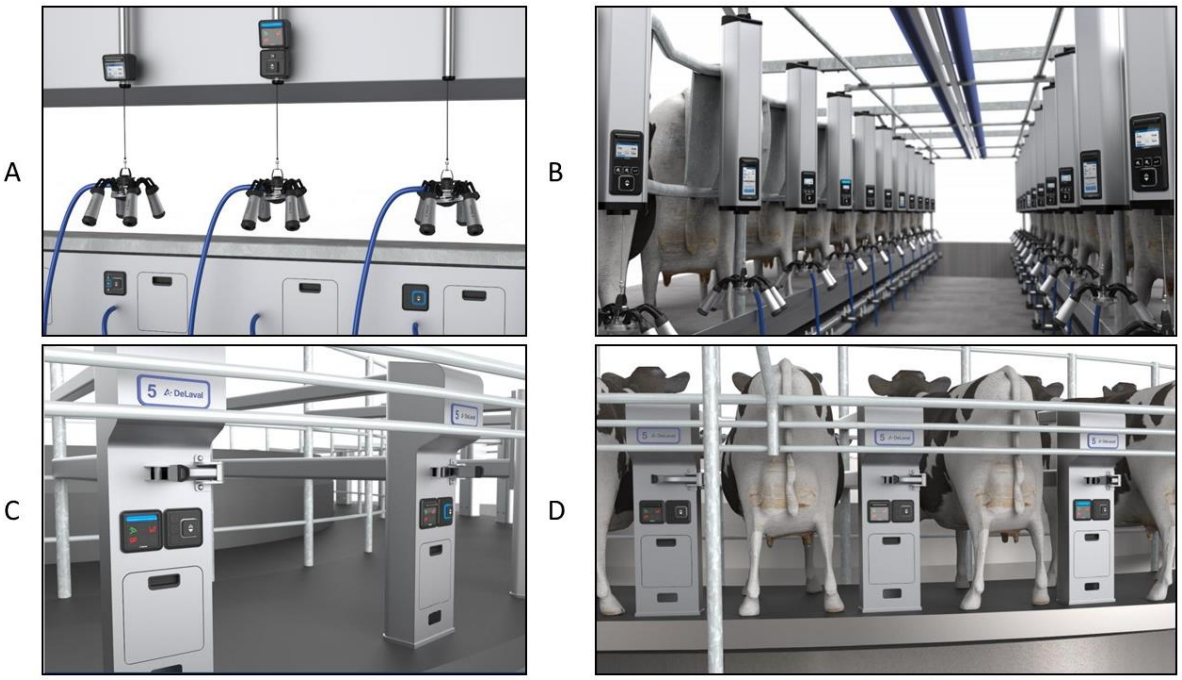
Systemen som är inringade med heldragen vit linje och numrerad med en 2a är två CCM-system (Control and Communication system). Dessa system hanterar kommunikationen mellan enheterna i sitt LIN-nätverk och kan kommunicera utanför nätverk genom Ethernet. Alla enheter som är streckat inringade i figur 2.3 utgör tillsammans ett LIN-nätverk och det tillhörande CCM-systemet är placerad nedanför.

2.2 DeLaval IOM 200

Under detta avsnitt kommer det inbyggda systemet IOM 200 från DeLaval att beskrivas. Först skildras systemets syfte och funktion, för att sedan ge en inblick av systemets hårdvara. Avslutningsvis beskrivs det inbyggda systemets mjukvara, kodstruktur och det realtidsoperativsystem som används i IOM 200.

2.2.1 Syfte och funktion

Systemet existerar i två olika stalltyper hos DeLaval som kallas mjölkningstall (engelska parlours) och roterande mjölkningstall (engelska rotarys). I ett mjölkningsstall mjölkas korna i gångar [4] och i ett roterande mjölkningstall går korna in i en mjölkningsplats på en roterande plattform [5]. I figur 2.4 finns fyra animerade bilder på mjölkstall, där de två första bilderna A och B är av vanliga mjölkningsstall och bilderna C och D är roterande mjölkningsstall.



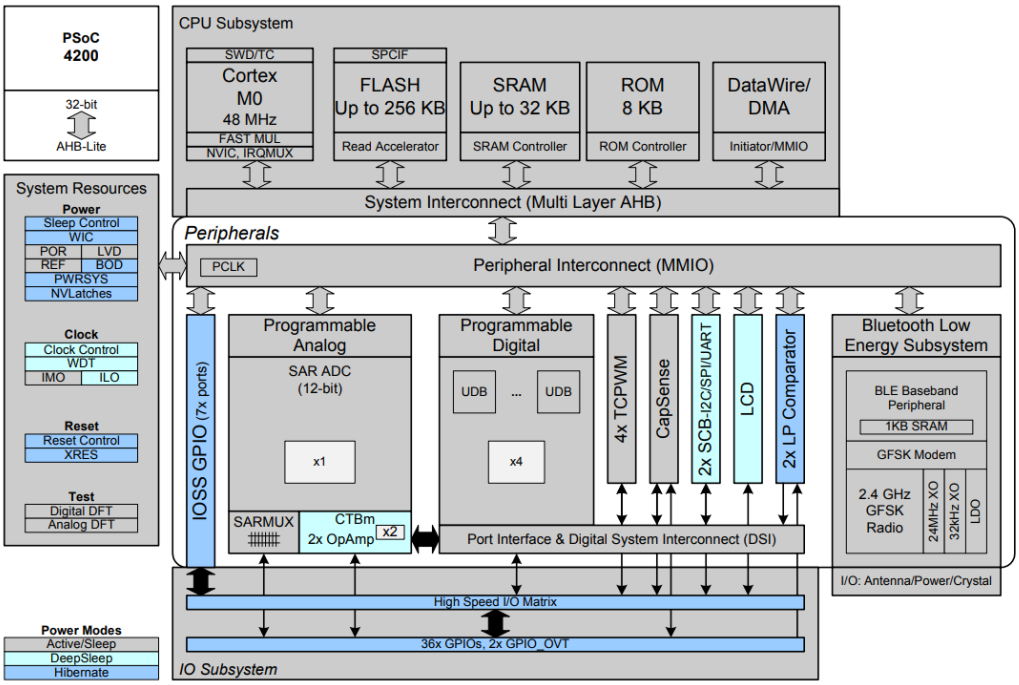
Figur 2.4. Animerade bilder på mjölkstall. De två första bilderna A och B är på mjölkningstall och bild C och D är ett roterande mjölkningsstall.

IOM 200 är ett inbyggt system som kallas för ett input/output (IO) system då systemets huvudsakliga uppgift är att hantera funktioner hos en eller flera mjölkplatser. IOM 200 har inget användargränssnitt, så som en skärm eller knappar då den inte ska ta emot information direkt från användaren. Det finns i stället en annan enhet som kallas MPC (Milking Point Controller) som är bondens gränssnitt för att styra mjölkningssystemet. MPC-enheten som skickar information till IOM 200 med instruktioner om vad som ska göras. IOM 200 kan även skicka information till andra enheter, den informerar exempelvis displayenhet DM 223 med statusen på mjölkflödet. All kommunikation med andra system (exempelvis MPC eller DM 223) sker över en LIN-buss och kommunikation med andra enheter sker genom in och utgångar hos IOM 200 systemet. Mer om kommunikationsgränssnitten i IOM 200 finns i avsnitt 2.2.4.

En IOM 200 hanterar funktioner för en mjölkplats i ett roterande mjölkningsstall. Systemet kan stänga och ta av klustret från kon samt hantera trycket hos pulsatorn. Ett kluster är det som sitter på kons spenar när den mjölkas. Ett kluster består av fyra spenkoppar, en för varje spene på kons juver. Pulsatorn är en enhet som alternerar vakuum för att skapa ett tryck i spenkopparna och på så sätt klämma runt spenarna. Systemet sköter även kontakten med sensorn för mjölkflödesmätning under mjölkning och med aktuatorerna som hanterar bommen som ger stöd åt kon på plattformen. En IOM 200 i ett mjölkningsstall kan hantera funktioner i upp till fem mjölkplatser. Systemet sköter om klustret och den har direkt kontakt med de ställdon som klustret hänger i.

2.2.2 Mikrokontroller från Cypress

I det inbyggda systemet IOM 200 sitter en mikrokontroller från Cypress av deras familjetyp PSoC 4: 4200_BLE, den exakta modellbeteckningen är CY8C4248LQI-BL483 (paketet QFN). PSoC 4 är en akronym för Programmable System-on-Chip, och är en plattformsarkitektur för programmerbara inbyggda system som använder en ARM Cortex-M0 processor. Mikrokontrollen har en integrerad Bluetooth Low Energy (BLE) modul och den stödjer Bluetooth 4.1 [6]. I figur 2.5 finns ett blockdiagram över mikrokontrollen, diagrammet är taget från mikrokontrollens datablad från Cypress.



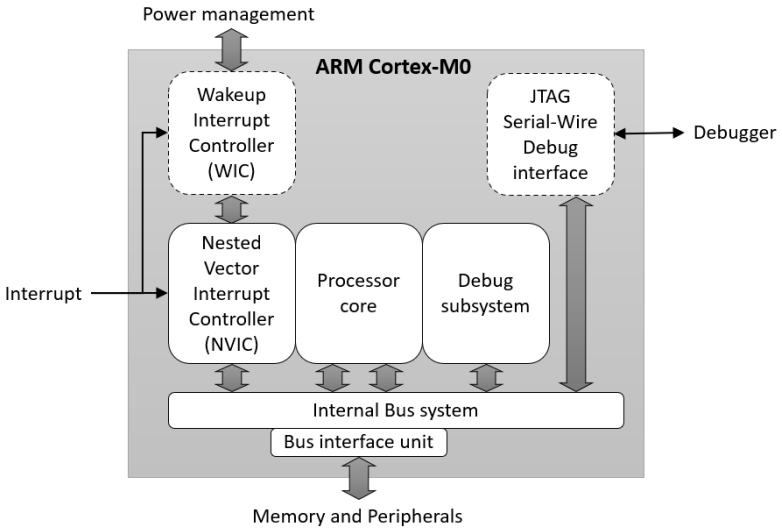
Figur 2.5. Blockdiagram över PSoC® 4: 4200_BLE mikrokontroller från Cypress [6].

Mikrokontrollen stödjer flertalet kommunikationsgränssnitt, så som olika typer av general purpose input/output (GPIO) och seriella gränssnitt (SPI/I2C/UART). De olika gränssnitten hos mikrokontrollen illustreras i Figur 2.5 och de gränssnitt som används i systemet IOM 200 beskrivs i avsnittet 2.2.4 Kommunikationsgränssnitt.

2.2.3 ARM Cortex-M0 processor

Programvaran för ett inbyggt system är designat för att exekveras av den specifika processor som finns i systemet. Det innebär att processen och logiken i programmet är designat utefter den specifika processorns funktionalitet. När en processor exekverar ett program måste den läsa in maskininstruktioner från primärminnet till processorns register för att sedan utföra uppgifterna. Processorns funktion kan beskrivs i tre steg, först hämtas programmet ur maskininminnet, därefter avkodas programmet (adresshantering) för att avslutningsvis utföra instruktionerna [7]. Det finns olika typer av processorer och de kan arbeta och hantera processorfunktioner olika. Därför behöver ett program designat för en specifik processor exekveras på den processorn eller i en miljö som fungerar på samma sätt.

Processorn som finns i det inbyggda systemet DeLaval IOM 200 är en ARM Cortex-M0 processor. Det är en 32-bitars Reduced Instruction Set Computing (RISC) processor som använder Thumb-2 som instruktionsset. Thumb-2 teknologin stödjer både 16 och 32-bitars instruktioner och tillåter samtliga instruktioner att exekveras utan att byta tillstånd i processorn. 32-bitars instruktioner utnyttjas då en 16-bitars instruktion inte kan utföra hela operationen [8].



Figur 2.6. Enkelt blockdiagram över en ARM Cortex-M0 processor.

I figur 2.6 illustreras en förenklad Cortex-M0-processor som finns i DeLaval IOM 200 i form av ett blockdiagram. I mitten av processorn finns processorns kärna tillsammans med ett debuggsystem och nested vector interrupt controller (NVIC). Processorkärnan är den delen av enheten som hanterar processorns trestegsprocess för att hämta, avkoda och exekvera instruktioner. Processorkärnan innehåller register, datavägar, kontrollogik och en aritmetisk logisk enhet (ALU, är en räkneenhet som utför logiska operationer) [8].

NVIC är en kontrollerenhet för processorns avbrottshantering. Denna enhet tar emot alla avbrottförfrågningar och avbrotten måste godkännas utav NVIC-enheten innan processorkärnan får information om den. NVIC har funktioner för att den kan jämföra och prioritera mellan avbrott om flera avbrott vill exekvera samtidigt. Den NVIC som sitter i ARM Cortex-M0 accepterar upp till 32 avbrottsförfrågningar [8].

Debuggsystemet kan kontrollera och hantera alla debuggfunktioner hos processorn. Debuggsystemet kan stoppa processorn och sätta den i ett stoppat tillstånd om ett debuggevent inträffar, kan exempelvis vara en brytpunkt. I ett stoppat tillstånd kan utvecklaren undersöka processorns aktuella status [8].

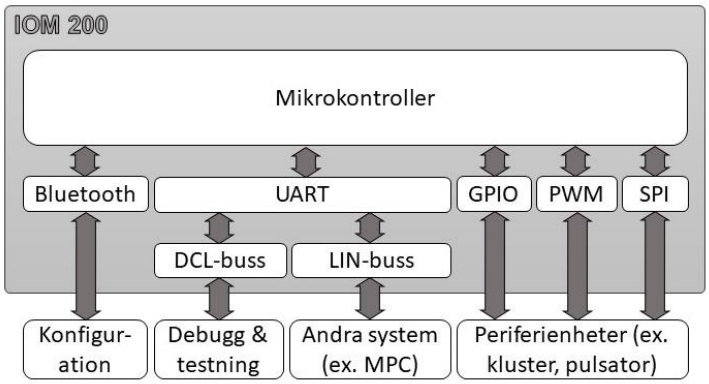
Det interna bussystemet och bussgränssnittet har båda en bredd på 32-bitar och möjliggör kommunikation mellan processorenheter, periferienheter och minnet. JTAG är ett gränssnitt och protokoll som tillåter åtkomst till processorn och dess debuggfunktioner [8].

Wakeup interrupt controller (WIC) är en valfri enhet som kan detektera avbrott när systemet är i sovande tillstånd (standby state). När systemet är i ett sovande tillstånd är processorkärnan och NVIC inaktiva, om ett avbrott sker informerar WIC spänningsövervakningen (power management) att det

är dags att starta systemet. På så sätt aktiveras NVIC och processorkärnan så att avbrottet kan hanteras [8].

2.2.4 Kommunikationsgränssnitt

Det inbyggda systemet IOM 200 har flera uppgifter som kräver extern kommunikation. Med det menas att IOM 200 måste kommunicera med enheter och system som ligger utanför det egna systemet (utanför den låda som omsluter IOM 200). Mikrokontrollen som finns i IOM 200 stödjer flera kommunikationsgränssnitt som exempelvis GPIO och SPI [9]. I figur 2.7 nedan illustreras de kommunikationsgränssnitt som används i IOM 200 för systemets olika funktioner.



Figur 2.7. Blockdiagram över kommunikationsgränssnitten från mikrokontrollen till externa enheter och funktioner i systemet IOM 200.

Bluetooth är en global standard för trådlös kommunikation som har existerat i över två decennier [10]. Bluetoothgränssnittet ingår i mikrokontrollen CY8C4248LQI-BL483 från Cypress som finns i IOM 200. Den stödjer Bluetooth 4.1 som också kallas Bluetooth Low Energy (BLE) [6].

Universal Asynchronous Reciever/Transmitter (UART) är ett asynkront och seriellt kommunikationsgränssnitt [9]. Det används två separata UART-gränssnitt i IOM 200, ett för kommunikation på en DCL-buss och en annan för en LIN-buss. DCL-bussen är ett gränssnitt som används utav utvecklare och tekniker på DeLaval för att simulera tester, skicka fabrikerade värden och debugga systemen. LIN-bussen är det kommunikationsgränssnitt som används mellan alla system i nätverket. Så om IOM 200 exempelvis tar emot information från MPC-systemet gör den det på LIN-bussen.

För att instruera och kommunicera med de externa enheterna som IOM 200 ansvarar över finns det tre olika gränssnitt som systemet använder GPIO, PWM och SPI. De två första gränssnitten är ut och insignaler på en pinne och den tredje (SPI) är seriell kommunikation.

General Purpose Input/Output (GPIO) är ett kommunikationsgränssnitt som sker på en pinne. Pinnarna kan konfigureras olika och deras funktioner finns beskrivet i databladet för mikrokontrollen. I detta fall med mikrokontrollen från Cypress så har alla GPIO's pinnar samma konfigurationsmöjligheter. De finns åtta olika körlägen på GPIO-pinnarna, de kan exempelvis vara pull-up/pull-down, analog/digital eller open drain [9].

Pulse Width Modulation (PWM) är en utsignal där signalen skickas i pulser med en viss period. Pulsens bredd beskrivs som procentuell mot signalens period och kallas även ”duty cycle” på engelska

(period menas hur ofta en puls upprepas, exempelvis en gång i sekunden). En pulsbredd på 25% innebär att en puls varar i en fjärdedel (1/4) av periodtiden, exempelvis en puls på 100µs med en periodtid på 400µs [11].

Serial Peripheral Interface (SPI) är ett synkront seriellt kommunikationsgränssnitt. För kommunikation med SPI måste enheterna vara konfigurerade som ”master” eller ”slave”. Kommunikation kan bara ske mellan en master och en slave, där det är mastern som startar kommunikationen. För SPI kommunikationen i IOM 200 är mikrokontrollen mastern-enheten och den stödjer kommunikation med flera slave-enheter [9]. Ett exempel på en slave-enhet som IOM 200 kommunicerar med är en sensor för mjölkflödesmätning.

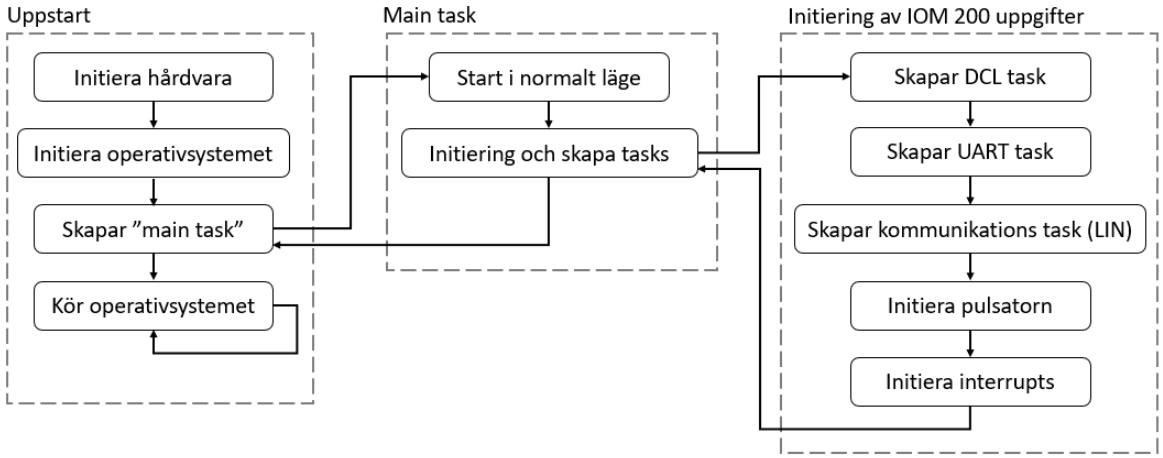
2.2.5 Mjukvara och kodstruktur

All kod för det inbyggda systemet IOM 200 finns i ett stort bibliotek som också förvarar annan kod som berör andra system. Det stora bibliotek är systematiskt uppdelat i sektioner (mappar) som är kategoriserade enligt funktioner. I dessa sektioner samlas alla bibliotek, wrappers, testfiler, kodfiler och liknande som berör allt i kategorin som är sektionen. Vissa sektioner har innehåll som används av flera system medan andra sektioner är specifika för ett inbyggt system.

DeLaval har flertalet wrappers för olika gränssnitt, det finns exempelvis wrappers för kommunikation och operativsystem (FreeRTOS).

I det inbyggda systemet IOM 200 används operativsystemet FreeRTOS. Det är ett realtidsoperativsystem och det ansvarar för systemets resurser och exekverar systemkod. Alla funktioner som ska upprepas är placerade i tasks (uppgifter) enligt FreeRTOS standard. Dessa uppgifter schemaläggs och prioriteras (enligt inställningar) av FreeRTOS kernel under körning. Mer information om FreeRTOS finns i nästa avsnitt.

I figur 2.8 illustreras en förenklad process över IOM 200s programkod. När IOM 200 startas så initieras först hårdvara och operativsystem. Där efter initieras och skapas alla uppgifter (tasks) för IOM 200. Kommunikationsuppgiften är den task som hanterar kommunikationen på LIN-bussen. Initieringen av pulsatorn och avbrott (interrupts) är enkla och är inte uppgifter för FreeRTOS. När alla förberedelser är klara startas operativsystemet FreeRTOS och den kör programmet tills att systemet stängs av.



Figur 2.8. Förenklad illustration över processen för programkoden i IOM 200.

2.2.6 Realtidsoperativsystem FreeRTOS

I det inbyggda systemet IOM 200 används ett realtidsoperativsystem (RTOS) för att köra programvaran. När ett program körs i ett inbyggt system kan det köras i vad som kallas en superloop. I den här loopen exekveras alla uppgifter succesivt från toppen till botten för att sedan göra samma sak igen, för alltid [12]. Ett annat sätt är att använda ett operativsystem som stödjer exekvering av flera program parallellt. Att köra och hantera flera program simultant kallas för ”multitasking” då dessa program ofta kallas för ett ”task” (uppgift). En processorkärna kan bara köra en tråd åt gången, det löser operativsystemet genom att schemalägga uppgifterna som ska utföras, de får med andra ord dela på processorkapacitet. För operativsystemet finns en så kallad schemaläggare som bestämmer vilken uppgift som får exekveras. Denna schemaläggare kan prioritera uppgifter och avbryta en uppgift som körs för att exekvera en med högre prioritet. Om schemaläggaren avbryter en uppgift har den koll på att avbrottet får ske och att det görs på ett bra sätt, den ser också till att det återupptas när de med högre prioritet är klart. Genom schemaläggandet av uppgifter och de snabba bytena mellan uppgifter så upplevs det som om programmen körs simultant [13].

Termen ”realtid” i RTOS syftar på att tiden för uppgifter som utförs enligt ett schema är förutsägbart eller den term som oftast används för att beskriva det är ”deterministisk” (deterministic). Realtid syftar inte till att vara snabb utan att de uppgifter som ska utföras av systemet alltid möter sina deadlines [13].

I det inbyggda systemet IOM 200 används FreeRTOS som realtidsoperativsystem. FreeRTOS är ett marknadsledande realtidsoperativsystem som fungerar för flertalet plattformar [13]. FreeRTOS är gratis att använda (open source) även i kommersiellt syfte och har utvecklats i över 18 år nu (2021). FreeRTOS är dessutom kostnadsfritt och väldokumenterat, med ett stort bibliotek och tillhandahåller stöd [13].

2.3 Linux

Utveckling och arbete med programkoden i IOM 200 sker idag (2021) på DeLaval i en Linuxmiljö. Linux är ett operativsystem (OS) och ett OS uppgift är att möjliggöra och ansvara över kommunikationen mellan datorns hårdvara och mjukvara. Operativsystemets ansvar ligger i att orientera och distribuera datorns olika resurser, så som minnet, för de funktioner som ska utföras av datorn. Operativsystemets kärna kallas för en ”kernel” och det är den enheten som styr operativsystemet [14].

På DeLaval används en Linux-distributionen Ubuntu 18.04. Arbetet har använt samma version av Ubuntu i en så kallad virtuell maskin. En virtuell maskin innebär att det körs ett datorsystem från ett annat datorsystem (originalet) [14]. I det här fallet har en virtuell maskin för Linux (Ubuntu) kört på en Windowsdator (Windows 10). Ubuntu är en öppen programvara som är gratis att använda [15].

2.3.1 FreeRTOS/POSIX simulator för Linux

FreeRTOS erbjuder en port för att köra en simulering av deras operativsystem i Linux. För att porta FreeRTOS och simulera uppgiftshanteringen (multitasking) används POSIX-trådar [16]. POSIX är en internationell standard från IEEE som står för Portable Operating System Interface for Computing Enviroment. POSIX-standarden definierar hur applikationer ska hantera operativsystemets funktioner. Genom POSIX kan program enklare flyttas mellan olika operativsystem, som använder POSIX [17]. Denna FreeRTOS-simulator använder POSIX biblioteket som finns i Linux och tillhandahåller en POSIX-wrapper för FreeRTOS plattformen [16].

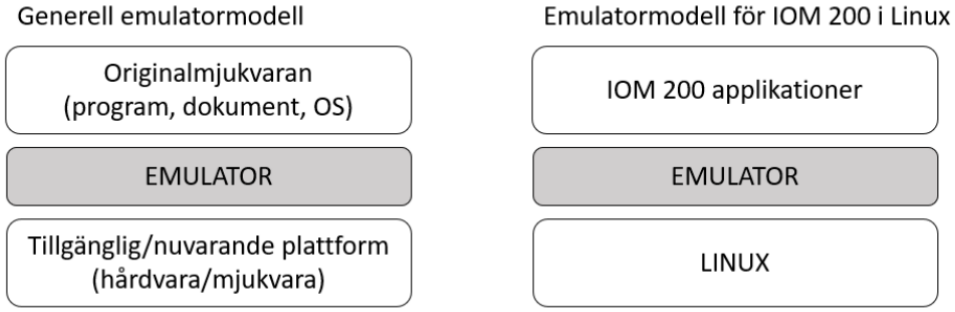
FreeRTOS-simulatorn kommer inte att uppvisa ett realtidsbeteende. Simulatorn är en plattform som visualiserar och agerar som ett FreeRTOS operativsystemet gör i ett inbyggt system, men utan kontroll på realtiden. Simulatorn kommer att ha samma deterministiska skiftningar mellan uppgifter (tasks) som i ett riktigt system [16].

2.4 Emulering och simulering av inbyggda system

Detta avsnitt börjar med att förklara vad en emulator är och hur en emulering fungerar och används. Därefter beskrivs de abstraktionsnivåer en emulator kan uppnå vid emulering av ett inbyggt system. Avslutningsvis presenteras befintliga arbeten där de emulerat hela eller delar av ett inbyggt system.

2.4.1 Emulatormodellen

En emulering är en imitation av ett specifikt system på/i ett annat system. Ett system kan vara ett datorprogram eller en plattform. En emulator är ett program som tillhandahåller ett gränssnitt mellan värdeplattformen och mjukvaran från originalplattformen. Emulatorgränssnittet möjliggör exekveringen av applikationer från originalsystemet på en alternativ plattform [18]. I figur 2.9 illustreras den generella emulatormodellen tillsammans med emulatormodellen för det inbyggda systemet IOM 200 om den skulle emuleras på en Linuxplattform.



Figur 2.9. Illustration av den generella emulatormodellen och emulatormodellen för det inbyggda systemet IOM 200 i Linux.

2.4.2 Abstraktionsnivåer hos en emulator

Det finns abstraktionsnivåer att ta i beaktning vid utformningen av den plattform som ska emulera hårdvaran i ett inbyggt system. Vilken abstraktionsnivå som emuleringen ska ligga på är beroende av syftet som plattformen ska uppfylla, alltså vilken funktion den ska uppfylla och användas till. Abstraktionsnivåerna finns listade i tabell 2.1 och djupare beskrivningar kommer i styckena nedan [19].

Tabell 2.1: Abstraktionsnivåer för emulering av ett inbyggt system.

Nivå	Abstraktionstyp	Beskrivning
Högst	Beteende	Utvecklarens perspektiv, utan tidsanpassning
Hög	tidsanpassat	Utvecklarens perspektiv med tidsanpassning, vag och approximerad tidtagning, approximerade cykler
Låg	Cykelanpassat	korrekta cykler och klockning
Lägst	Implementationsanpassat	Korrekta cykler, klockning och registerhantering. Register-transfer level (RTL), design simulation model (DSM)

Den högsta graden av abstraktionsnivå är ett program som efterliknar ett systems beteende, ur en utvecklares perspektiv utan någon hänsyn till systemets eller programmets tidsaspekter. Det betyder

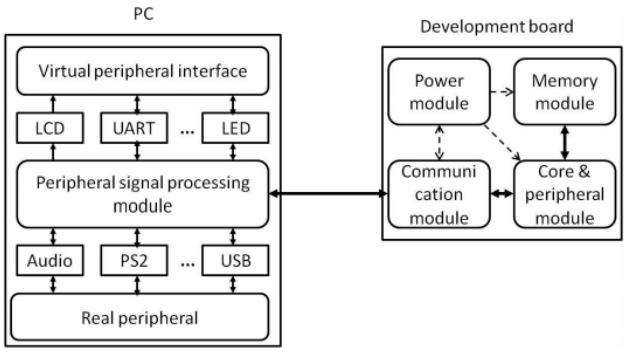
att en emulering med högsta abstraktionsnivån ska simulera ett systems funktioner korrekt på så sätt att emulatorns beteende överensstämmer med det riktiga systemet [19].

I den höga abstraktionsnivån simulerar programmet systemets beteende samtidigt som den har tidsanpassats. En emulator i den höga nivån kan vara mer eller mindre tidsanpassad, det räcker med att någon del av programmet har tagit en tidsaspekt i beaktning för att flyttas ner från den högsta abstraktionsnivån. Ett program kan starta i den högsta abstraktionsnivån för att sedan övergå till den tidsanpassad abstraktionsnivån genom att succesivt utveckla delar av programmet för att möta systemets tidsaspekter [19].

I den låga abstraktionsnivån har programmet gått steget längre än tidsanpassning och kan nu simulera korrekta cykler och klockning som överensstämmer med det som sker i det riktiga systemet. I nästa abstraktionsnivå (som är den lägsta) så simuleras även systemets hårdvarugränssnitt så som registerhantering, minnesallokering, hårdvaruinstruktioner etcetera [19].

2.4.3 Virtuella periferienheter

I ett annat arbete skapades virtuella periferienheter till ett utvecklingskort som ska användas i utbildningssyfte. Arbetets syfte var att minska kostnaden och öka flexibiliteten för utvecklingskortet genom de virtuella periferienheterna. Konceptet för det arbetet var att behålla kärnan av utvecklingskortet och sedan skapa ett virtuellt gränssnitt för periferienheterna [20]. I figur 2.10 illustreras konceptet för deras arbete.



Figur 2.10. Koncept för ett utvecklingskort med ett virtuellt periferienhetsgränssnitt [20].

Som det illustreras i diagrammet i figur 2.10 så skickas kommunikationen med periferienheterna från utvecklingskortet till datorn för att tas emot av en modul för att ”processa periferisignaler”. Från den modulen illustreras olika kommunikationsgränssnitt som utvecklingskortet kan tänkas använda. Vissa periferienheter kan utnyttja de som finns tillgängligt på datorn, så som ljud eller USB. De periferienheter som inte finns tillgängligt från datorn ersätts i det virtuella gränssnittet [20].

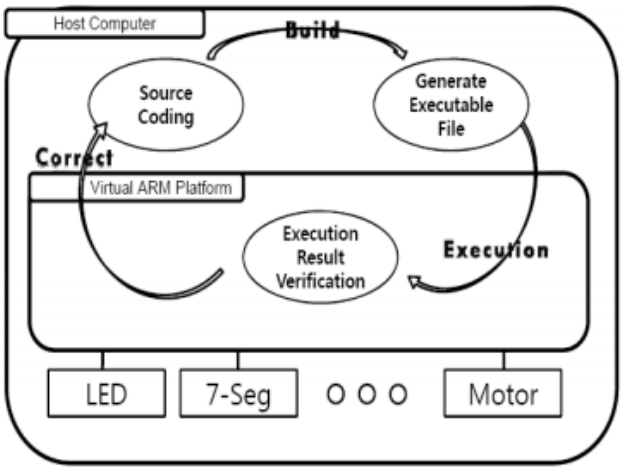
Arbetet testade sitt koncept mot utvecklingskortet LEON3 GR-XC3S-1500. Utvecklingskortets kärna och periferienhetsmodul (se figur 2.10) kommunicerar antingen med kortets minne eller med kommunikationsmodulen. I vanligt fall hade kommunikationen till periferienheter gått från kortet till periferienhetskontakter, men arbetet dirigerar i stället om signalerna till datorn med hjälp av en wrapper-funktion. Wrapper-funktionen paketerar signalerna så att datorn kan hantera dem. Funktionen kan också översätta insignaler från det virtuella gränssnittet till signaler kortet kan förstå [20].

Det virtuella periferigränssnittet fick ett grafiskt användargränssnitt där periferienheterna visualiserades. Användargränssnitt blev ett fönster med text, figurer och knappar som skulle illustrera olika

periferienheters statusar samt möjliggöra inmatning från användaren. LED-lampor visualiserades exempelvis med gröna cirklar, där en lysande LED illustrerades genom färgen neongrön och en släckt hade färgen mörkgrön [20].

2.4.4 Virtuell ARM-plattform

Det finns flertalet simulatorer för ARM-processorer, så som exempelvis ARMulator och SimIt-ARM. ARMulator erbjuder en virtuellmiljö för mjukvaruutveckling för ett inbyggt system utan dess hårdvaruplattform. Det åstadkommer ARMulator genom att virtuellt implementera protokollet för hårdvarugränssnittet och genom att porta operativsystemet. SimIt-ARM är en simulator som simulerar instruktionssetet och kan köra ARM program på både system och användarnivå. SimIT-ARM måste länka kontrollkoden för in och utsignaler med C Runtime bibloteket (CRT) för att skapa en exekverbar kod, medan länkningen av signalerna skulle ske mot uppstartskoden på hårdvaran. Det betyder att samma kod som simuleras inte kan exekveras på hårdvaran [21].



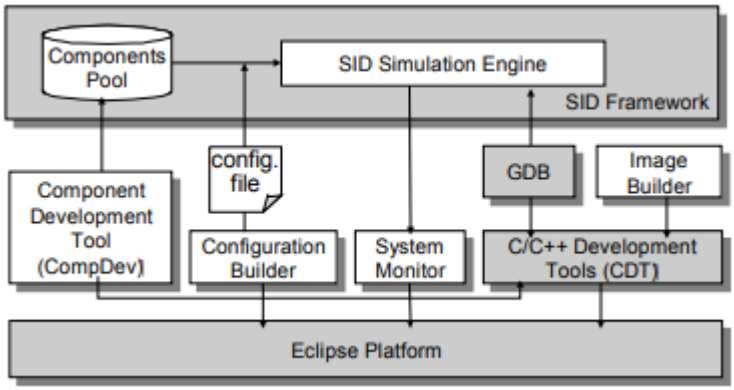
Figur 2.11. Processen för utveckling av mjukvara mot en virtuell ARM plattform [21].

I en konferenshandling publicerad i IEEE presenteras en virtuell ARM plattform som har ett grafiskt användargränssnitt (GUI), insignalshantering, timer och periferienheter. Konceptet och processen för att jobba med den virtuella plattformen illustreras i figur 2.11 som är taget ur konferenshandlingen. Plattformen baserades på en befintlig ARM simulator som arbetet byggde ut för att hantera periferienheter och för att den kod som exekveras i den virtuella plattformen ska fungera i den riktiga hårdvaran [21].

När plattformen används börjar det med att skapa och presentera det grafiska användargränssnittet. Sedan körs ARM simulatorn, den processar instruktionerna i den kompilerade filen och skickar resultatet till plattformen. När plattformen tar emot resultaten så uppdaterar den det grafiska gränssnittet för att illustrera statusen. Om en insignal kommer analyseras det av plattformen för att sedan skickas till ARM simulatorn för att utföra instruktionerna [21].

2.4.5 VDEES

I ett arbete vars rapport publicerats i IEEE Xplore har man skapat en virtuell utvecklingsmiljö för inbyggd mjukvara, som de kallar VDEES (akronym för virtual development enviroment for embedded software). Arbetet resulterade i en virtuell plattform (VDEES) som kan anpassas mot en specifik hårdvara och som möjliggjorde utvecklingen av mjukvara för ett inbyggt system på en virtuell plattform. I figur 2.12 så illustreras arkitekturen hos den virtuella plattformen, taget ur arbetets rapport i IEEE [22].



Figur 2.12. VDEES arkitektur [22].

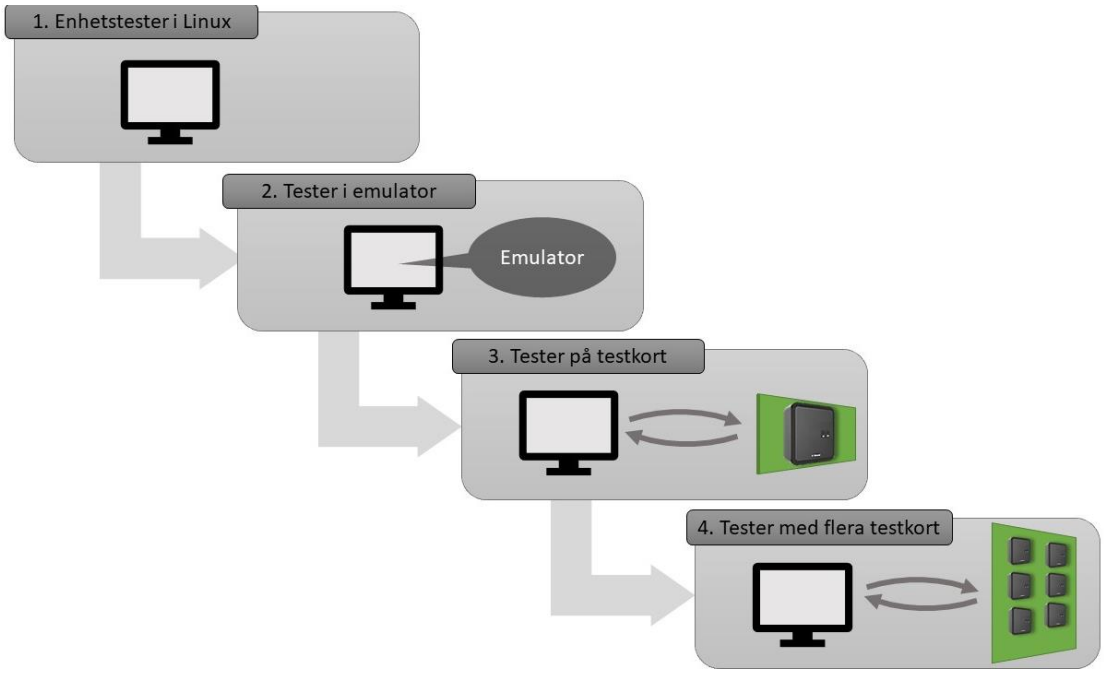
Arbetet använde Eclipse som utvecklingsplattform och SID som simuleringsverktyg/maskin, båda programmen är gratis att använda. C/C++ bibliotek (CDT) är ett tillbehör från Eclipse IDE, den innehåller bland annat en editor, kompilator, länkare och debugger. Arbetet valde att utöka CDT med det som i figuren kallas CompDev och Image Builder. Det som utvecklas i CDT kan bara köras på samma plattform som den utvecklades på. Denna problematik skall Image Builder modulen lösa. Modulen erbjuder en miljö som kan stödja flera plattformar och den har en ”general public licens” (GNU). För simuleringen behövs en konfigurationsfil som ska beskriva målplattformen, de olika komponenterna och deras relationer. För att göra konfigureringen lättare för användaren skapade arbetet ett grafiskt användargränssnitt (GUI) för att fylla i informationen. SID har ett inbyggt system för att övervaka körande simuleringar, men detta verktyg är enligt arbetet både begränsat och komplicerat. Eclipse stödjer bara Java plug-ins (program tillbehör) medan SID verktyget inte kan ta emot java komponenter direkt. Detta problem löser arbetet genom ett använda ett kommunikationsuttag för ändamålet. Modulen CompDev är den enhet som möjliggör utvecklingen av nya SID komponenter, även här finns ett GUI för att lägga till komponenter. När nya komponenter skapats genom GUI skapas det en tom huvudklass (main) och tillsammans med alla konfigurationsfiler för komponenten. När kodningen sedan är klar så samlar en parser all information och skriver det i en xml-fil som SID verktyget kan använda [22].

3 Metoder och resultat

Detta kapitel börjar med att presenteras konceptet, modellen och utvecklingsprocessen för byggandet av emulatorn. Efter den teoretiska inledningen beskrivs den resulterade emulatorprototypen som utvecklats under arbetet.

3.1 Koncept

Syftet med emulatorn var att underlätta i testprocessen på DeLaval för deras inbyggda system IOM 200. Testprocessen som den ser ut idag finns beskrivet i avsnitt 2.1 tillsammans med en illustration i figur 2.1. Konceptet var att implementera en emulator som möjliggör testandet av beteendet i koden hos IOM 200. Implementationen av denna emulator skulle ändra testprocessen genom att lägga till ett steg mellan enhetstesterna i Linux och testandet på ett testkort. Den altererade testprocessen med emulatorkonceptet illustreras i figur 3.1



Figur 3.1. Testprocessen för koden i IOM 200 med emulatorkonceptet.

Emulatorkonceptet minskar omfattningen av den kod som måste testas på testkortet genom att möjliggöra fler tester i Linux. Testerna i emulatorn kan fokusera på validering av applikationens beteende då den inte beaktar hårvarugränssnitten och då enhetstesterna redan har säkerställt funktionen hos kodsegmenten. Om ett test genererar dåliga resultat eller oönskat beteende är det lättare att felsöka om omfattningen på koden som måste beaktas är mindre.

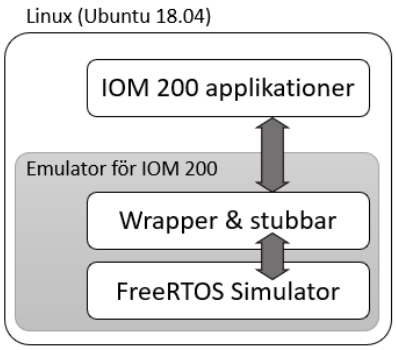
Emulatorn är ett program som kör och använder IOM 200 applikationer. Allt IOM 200 material finns samlat i DeLavals stora bibliotek tillsammans med material för andra inbyggda system. Det är därför logiskt att placera emulatorapplikationen här i DeLaval-biblioteket tillsammans med de resterande applikationerna. Det implementerades genom att samla allt material som berör eller utgör emulatorn har i ett paket och det ladesi sin tur in i DeLaval-biblioteket. Det samlade emulatormaterialet är olika typer av filer och bibliotek som tillsammans bygger emulatorapplikationen.

3.2 Emulatormodellen

Emulatorn har utvecklats i Linuxmiljön av typen Ubuntu med versionen 18.04 och skapades i utvecklingsverktyget Eclipse. Dessa valdes med avsikten att använda samma miljöer som utvecklarna på DeLaval använder vid utvecklingen programvaran för de inbyggda systemen. På så sätt samlas all utveckling och testning i samma miljö på datorn.

FreeRTOS-simulatorn för Linux fick stå som grund för den emulatorplattform som utvecklades under arbetet. Motiveringen för att använda denna simulator grundas i att FreeRTOS är det operativsystem som används i IOM 200. Ytterligare motivering till FreeRTOS-simulatorn är att den har den abstraktionsnivå av systemet som begärs av emulatorplattformen. Emulatorns syfte var att realisera beteendet hos applikationen för IOM 200, vilket innebär den högsta nivån av abstraktion (se avsnitt 2.4.2 om abstraktionsnivåer).

Koden från IOM 200 ska vara exekverbar i emulatorn utan ändringar. För att uppfylla det kravet och den tänkta funktionen hos emulatormodellen så behövs det stubbar och wrappers för att möjliggöra exekveringen av IOM 200 kod på FreeRTOS-simulatorn. För ökad förståelse illustreras emulatormodellen i figur 3.2.



Figur 3.2. Enkel illustration över emulatormodellen.

För att studera applikationens beteende i emulatorn kan programmet debuggas och stegas igenom i utvecklingsmiljön Eclipse.

3.3 Utvecklingsprocess för emulatorprototypen

Utvecklandet av emulatorn började med att få FreeRTOS-simulatorn att fungera. När FreeRTOS-simulatorn fungerade började wrappers och stubbar att utvecklas för att möjliggöra exekveringen av kod från IOM 200. Arbetet behandlade små kodsegment i taget för att minska omfattningen och mängden kod som bearbetas samtidigt. Den iterativa utvecklingsprocessen såg ut som följande:

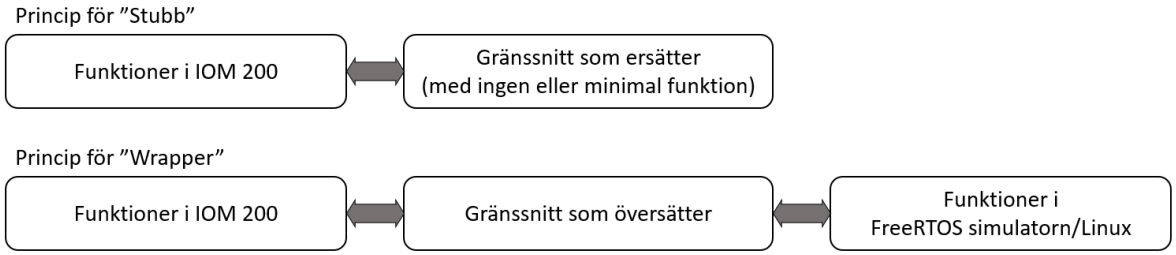
1. Väljer ut ett kodsegment från IOM 200.
2. Bygger ut, återanvänder eller skapar nya wrappers eller stubbar för att möjliggöra exekveringen i den emulerade miljön med FreeRTOS-simulatorn. Hårdvarugränssnitt stubbas bort och operativsystemsfunktioner behöver wrappers för att passa FreeRTOS-simulatorn. Funktioner som för arbetet inte är av intresse eller ligger utanför emulatorns stubbas bort.
3. Körningar och tester görs med kodsegmentet mot FreeRTOS-simulatorn med wrapper/stubbar. Detta steg och steg 2 kan pendla fram och tillbaka för att testa delar av segmentet under utvecklingen. När implementeringen av hela kodsegmentet är validerat börjar processen om från steg 1.

3.4 Stubbar och wrapper-funktioner

I emulatormodellen för IOM 200 som presenterades i avsnitt 3.2 så utgörs emulatorn av FreeRTOS-simulatorn och ett bibliotek med stubbar och wrapper-funktioner. Biblioteket innehåller alla filer som tillhandahåller de gränssnitt som möjliggör att en IOM 200 applikation kan exekveras på FreeRTOS-simulatorn.

En stubb är en funktion som ersätter en annan funktion men som inte gör någonting eller som alltid gör samma sak. I det här arbetet har funktioner som kommunicerar med hårdvaran, eller med något som inte finns eller fungerar i den virtuella miljön stubbats bort. För att en stubb-funktion skulle fungera så måste den ha samma namn som den som används i IOM 200 applikationen. Funktionen måste även ha samma inparametrar och returparametrar av rätt typ. Om funktionen ska returnera någonting kan detta returvärde vara statiskt om det inte uppfyller någon funktion för simulerandet av programmets beteende. Ett exempel på en stubb som arbetet skapat är för en funktion som i IOM 200 frågar efter kortets typ och ger ett artikelnummer som retur. Denna funktion är hårdvarubaserad och har därför ersatts med en funktion som alltid returnerar artikelnumret för IOM 200.

En wrapper har i det här arbetet inneburit en funktion som översätter en befintlig funktion i IOM 200 applikationen till en funktion som finns i emulatormiljön. Det har framför allt innefattat att översätta de operativsystemsfunktioner som finns i IOM 200 koden till deras motsvarighet som finns i FreeRTOS-simulatorn. Ett exempel på en wrapper-funktion som finns i den utvecklade emulatorn är en för skapandet av FreeRTOS-tasks. Funktionsnamnet för att skapa FreeRTOS -tasks i IOM 200 har ett annat namn än den som finns i FreeRTOS-simulatorn. Därför skapades en wrapper-funktion som hette det samma som den i IOM 200 och som tar de in parametrarna och anropar den funktion finns i FreeRTOS-simulatorn. I figur 3.3 som finns nedan illustreras principerna som arbetet utformats efter för att skapa gränssnitten till emulatorn.



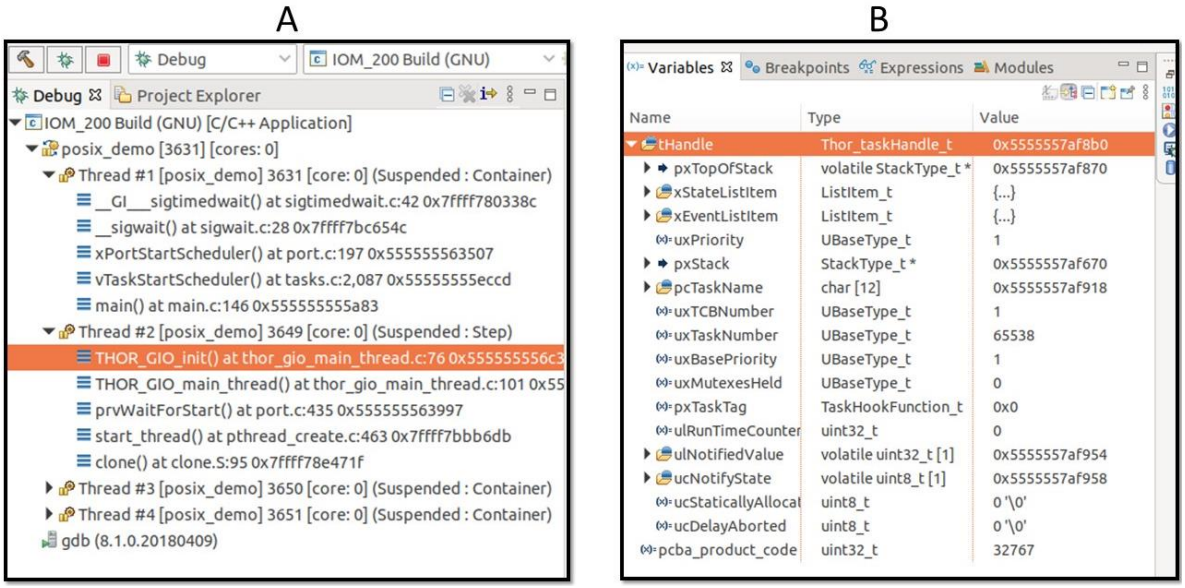
Figur 3.3. Diagram som beskriver principiellt hur arbetet använt och jobbat med stubbar och wrapper-funktioner.

Utöver funktioner så behövdes även en del variabler, konstanter och datatyper översättas och definieras. Exempelvis länkades namnet på datatypen för task-hantering i IOM 200 till den som finns i FreeRTOS-simulatorn (FreeRTOS datatyp: TaskHandle_t).

3.5 Emulatorprototypen

Den emulator som skapats under arbetet är en prototyp som realiserat emuleringen av ett mindre segment från IOM 200 applikationen. Emulatorn är byggd enligt den modell som presenterats tidigare i kapitlet (se även figur 3.2). I det tidigare avsnittet 3.4 presenterades hur gränssnittet mellan IOM 200 applikationen emulatorn bearbetats fram tillsammans med några praktiska exempel från emulatorprototypen. All kod som utgör wrapper och stubbgränssnittet har i emulatorprototypen samlats i två filer, en c-fil och en tillhörande h-fil.

Emulatorprototypen är skriven och finns tillgänglig i utvecklingsmiljön Eclipse IDE (i en Linux av typen Ubuntu 18.04). För att följa och studera exekveringen av IOM 200 applikationen kan man köra emulatorn i debugg-läge. När emulatorn körs i Debugg-läge är det möjligt att stega igenom exekveringsprocessen och på så sätt studera beteendet hos programmet.



Figur 3.4. Två skärmklipp från emulatorprototypen under en körning i debugg-läge i Eclipse. Skärmklipp A visar debugg-vyn och skärmklipp B visar variabelvyn.

I figur 3.4 ovan finns två skärmklipp från emulatorprototypen. I skärmklipp A visas hur debugg-vyn kan se ut när prototypen är stoppad under en körning. I debugg-vyn visualiseras stacken, operationer och de trådar som används. Emulatorn använder en GNU-baserad verktygskedja för kompilering och debugging. Namnet ”posix_demo” är själva testapplikationen. Ordet ”Thor” som finns i flertalet namn är referenser till IOM 200. I skärmklipp B visas variabelvyn som kan observeras under körningen i debugg-läge. Här kan man se deklarerade variabler och deras värden under en körning.

Kodsegmentet från IOM 200 som körs i emulatorn hanterar fyra FreeRTOS-uppgifter (tasks), dessa hanteras som trådar (threads) i FreeRTOS-simulatorn och syns i Skärmklipp A i figur 3.4. Programmet börjar med att skapa en huvuduppgift (Thread #1), den uppgiften skapar ytterligare tre andra uppgifter och startar slutligen schemaläggaren. När schemaläggaren (från FreeRTOS-simulatorn) startas tar den över programmet och kör de tre andra uppgifterna som är från IOM 200 applikationen. En uppgift tar emot information om statusen på klusterhanteringen, en stubbad funktion som inte gör något i prototypen. En annan uppgift är blinkandet av lysdioden som finns på systemets låda och som indikerar statusen på systemet under körning. För denna uppgift används en FreeRTOS-timer vars funktion också finns tillgänglig i FreeRTOS-simulatorn. Denna timer har ingen realtidsuppfattning eller punktlighet utan estimerar tiden i millisekunder genom att använda systemets ”system-tick”.

4 Analys och diskussion

Under detta avsnitt kommer det utförda arbetet att analyseras och diskuteras. Kapitlet inleds med en diskussion kring de metoder, modeller och verktyg som beaktats i det här arbetet. Här motiveras även de valda metoderna och argumentationer förs mot de alternativ som valdes bort. Diskussionen förs sedan vidare till arbetets syfte och en analys av emulatorns hållbarhet. Därefter analyseras resultatet av detta arbete för att sedan avsluta med sina rekommendationer för fortsatt arbete och studier av emulatorn.

4.1 Metoder och verktyg

Under arbetet och utvecklingen av emulatorn användes olika metoder och verktyg. Arbetet har bara fokuserat på att använda datormiljön Linux då det var ett förbestämt krav. Arbetet har heller inte övervägt att använda en annan utvecklingsmiljö än Eclipse som idag används på DeLaval för koden till ett inbyggt system. Att stanna kvar i Linux och Eclipse är även i linje med arbetets mål med att underlätta testprocessen då utvecklarna inte behöver byta miljö eller program när de går från utveckling till testning av kod. Ett byte av miljö eller program för att köra testerna i emulatorn är ett onödigt steg som inte underlättar testprocessen i samma grad som det gör utan ett byte.

Den höga abstraktionsnivån som önskas hos emulatorn har varit en stor faktor vid val av metoder och simuleringsverktyg som för arbetet. Målet med emulatorn var att den ska möjliggöra exekveringen av det inbyggda systemets kod på så sätt att det går att studera och testa beteendet. Det tillsammans med inga krav på tidsanpassningar gör att emulatorn i detta arbete inte behöver gå lägre än den högsta abstraktionsnivån.

I ett annat arbete har möjligheterna till en emulering och skapandet av en virtuell plattform av en ARM-processor studerats. Som det tidigare tydliggjorts i det här arbetet så är koden i ett inbyggt system specifikt utvecklad mot den processor som ska exekvera programmet. Det är därför viktigt att efterlikna processorn för att det ska gå att exekvera den men också för att uppnå samma beteende vid en simulation. Denna metod är därför tillämpningsbar i detta arbete och är ett alternativ som enligt studien skulle uppfylla emulatorns mål.

Arbetet valde att inte utveckla en emulering med en virtuell processor utan att jobba med den redan existerande simulatorn för operativsystemet FreeRTOS. Arbetet hittade många fördelar utöver abstraktionsnivån för att jobba med FreeRTOS-simulatorn. Simulatorn är gratis att använda och den utvecklas och underhålls utav FreeRTOS själva. FreeRTOS-simulatorn används mot många olika processorer vilket gör att simulatorn fungerar på alla system som använder FreeRTOS. Den största anledningen till att använda FreeRTOS-simulatorn är att det innebär mindre utveckling för arbetet att skapa emulatorn. Eftersom applikationen i IOM 200 körs av FreeRTOS så är den också byggd enligt den standarden och med alla uppgifter. Det innebär att utan FreeRTOS-simulatorn måste arbetet själva ändå skapa och utveckla ett program som kan köra och efterlikna FreeRTOS.

I arbetet om VDEES fick arbetet en god insikt i hur ett simulationsverktyg kan implementeras och användas mot utvecklingsmiljön Eclipse. Plattformens arkitektur i det arbetet har både varit en bra inspirationskälla för att utveckla den generella emulatormodellen till den slutgiltiga emulatormodellen för IOM 200 i det här arbetet. Vidare gav VDEES studien en ökad förståelse av utvecklingsmiljön Eclipse och de tillhörande verktygen, så som GDB och CDT.

I ett av arbetena som studerats har ett inbyggt systems periferienheter simuleras i ett grafiskt gränssnitt på datorn. Konceptet och metoden är tillämpningsbar för emulatorn i detta arbete men den implementeringen är inte nödvändig för att skapa en emulator med högsta abstraktionsnivån där beteendet ska simuleras.

De höga abstraktionsnivåerna hos emulatorer kan medföra vissa begränsningar vilket måste tas i beaktning före användning av en emulator. Emulatorer av högre grad kan inte uppvisa realtid och kan därför inte användas i syften där tiden måste vara mätbar och realistisk. Den höga graden av abstraktion begränsar möjliga simuleringar då det inte emulerar korrekta cykler, klockning eller någon registerhantering.

4.2 Emulatorns syfte och funktion

Arbetets syfte var att underlätta DeLavals testprocess för kod till ett inbyggt system genom att implementera en emulator. I emulatorkonceptet som presenterades i det här arbetet introduceras emulatorn som ett nytt steg i testprocessen. Det nya emulatorsteget ska ligga efter enhetstestningen men innan testerna på målkortet (testkortet). Emulatormodellen ska göra det möjligt att exekvera koden från IOM 200 på Linux och följa den i Debugg-läge i Eclipse för att studera applikationens funktionalitet och beteende. Emulatorprototypen som skapades under arbetet visar att emulatormodellen är en fungerande lösning. Prototypen som är byggd enligt modellen har realiserat en emulator som kan exekvera ett mindre kodsegment från IOM 200. Prototypen går att köra i debugg-läge i Eclipse vilket gör det möjligt att följa och studera funktionen och beteendet hos IOM 200 applikationen.

4.3 Hållbarhet och etik

Arbetet har observerat och beaktat fördelarna med att implementera en emulator för ett inbyggt system. Det blir en minskad stress att utveckla och leverera hårdvaran av systemet om det är möjligt att exekvera och köra koden oberoende av den. Möjligheten att utveckla systemets mjukvara innan tillgången till hårdvaran ger flera fördelar, exempelvis behöver inte utvecklarna av mjukvara vänta på en hårdvaruprototyp. Om man utvecklar mjukvaran innan hårdvaruprototypen skapas är det enklare och billigare att ändra den då det inte finns en gammal prototyp som måste kasseras. Det ger även en ökad flexibilitet vid utvecklingen av systemet då företaget inte behöver beakta avfallet, kostnaden och det extra arbetet det innebär med flera prototyper. Företaget behöver heller inte tillhandahålla flera prototyper om det finns flera utvecklare som jobbar med mjukvaran.

En emulator kan begränsa riskerna för skador på maskiner, egendom, människor och djur. Anledningen till det kan härledas till att tester sker i en emulerad miljö i stället för med de riktiga maskinerna och systemen. Dessa system kan vid ett haveri orsaka skador av olika omfattning på både egendom och levande ting. Det är därför fördelaktigt att förenkla och gynna att fler tester kan utföras innan det körs i de riktiga systemen och maskinerna.

Arbetet vill lyfta bekvämlighetsfaktorn en emulator tillhandahåller utvecklaren vid arbete med ett inbyggt system. Arbetet observerade att det både var lättare och gick snabbare att sätta sig in i applikationen och förstå hela processen när den var möjligt att följa exekveringen i emulatorn. Då jämför arbetet med sina tidigare erfarenheter av att sätta sig in i ny kod, som är skriven av någon annan i ett inbyggt system utan en emulator. En annan bekvämlighet är att utvecklaren inte är beroende av hårdvaran för att göra sitt jobb, utvecklaren behöver inte lägga tid eller tanke åt att planera efter tillgången till hårdvaran.

En emulator av ett inbyggt system har en inkluderande effekt då en utvecklare inte behöver begränsas av tillgången till hårdvaran. Den fysiska hårdvaran har en kostnad som kan begränsa tillgängligheten. Vidare har arbetet observerat att utvecklare som jobbar hemifrån har olika förutsättningar på grund av olika arbetsmiljöer. Att arbeta med den fysiska hårdvaran hemifrån är av olika belastning för personalen på grund av olika miljöfaktorer. Det kan handla om för lite utrymme att jobba på eller kanske djur och barn som intresserar sig för enheterna. Oavsett hur utvecklarens kontor ser ut så behöver de inte begränsas av hårdvaran vid tillgången till en emulator.

En nackdel med implementationen av en emulator är att det tar både tid och resurser från företaget att utveckla en emulatorplattform. Omfattningen av de resurser som behövs är mycket varierande och är olika från fall till fall då plattformen måste anpassas efter det inbyggda systemet. Därför måste företaget själva utreda omfattningen av arbetet och avgöra om fördelarna väger upp kostnaden.

4.4 Resultat

Resultatet är i enlighet med de mål och avgränsningar som sattes för arbetet. Emulatorkonceptet visar hur emulatorn kan implementeras i DeLavals testprocess, emulatorn introduceras som ett till steg i processen som kommer efter enhetstesterna men innan testkortet. Emulatormodellen beskriver hela lösningen för emulatorplattformen och hur den ska byggas upp och implementeras tillsammans med DeLavals andra applikationer. Emulatorprototypen har realiserat en mindre del av applikationen i IOM 200. Trots den lilla applikationen så visar prototypen att modellen fungerar, den kör kod med flera FreeRTOS-uppgifter på FreeRTOS-simulatorn genom ett gränssnitt utvecklat av arbetet.

4.5 Rekommendationer och fortsatt arbete

För fortsatt arbete med den här emulatorn behöver prototypen utvecklas. Flera kodsegment från IOM 200 behöver implementeras succesivt till dess att hela applikationen realiserats. Arbetet rekommenderar att fortsätta med samma metod där mindre kodsegment realiseras i taget. Arbetet var nöjd med metoden och tyckte den gav större kontroll över funktionaliteten då det hanterades mindre kod åt gången.

Emulatorns syfte att underlätta testandet av kod för ett inbyggt system behöver utredas vidare. Fortsatta arbeten bör undersöka vinsterna av att implementera en emulator för ett inbyggt system i en utvecklingsprocess både ur ett generellt perspektiv och för specifika fall.

Det här arbetet har avgränsat sig från att undersöka och realisera avbrottshanteringen i emulatorn. Därför bör fortsatta arbeten utreda hur avbrott kan emuleras och med vilka abstraktionsnivåer det kan implementeras i en emulator.

Vidare kan emulatorns möjligheter till generalisering undersökas. Emulatormodellen som presenterades under detta arbete är användbar för alla system som använder FreeRTOS och är ute efter samma abstraktionsnivå. Modellen går att generaliseras genom att byta ut blocket som representerar IOM 200 applikationen mot ”systemets applikation”. Där systemet syftar till det inbyggda system som ska emuleras, med kravet att det använder FreeRTOS.

På DeLaval finns det flera inbyggda system med både liknande och samma hårdvara som i IOM 200. Arbetet vill därför rekommendera DeLaval att bygga ut emulatorn för att täcka alla deras inbyggda system. Så länge systemet använder FreeRTOS så är den implementerbar.

5 Slutsatser

Resultatet av arbetet är i enlighet med de uppsatta målen och inom avgränsningarna för arbetet. Resultatet är tillfredställande då arbete har:

- Utvecklat ett koncept för implementeringen av en emulator i DeLavals testprocess för koden i ett inbyggt system.
- Undersökt flera verktyg, arbeten och metoder som i någon form berör emulering av ett inbyggt system, liknande DeLavals IOM 200. Studien gav tillsammans med den önskade abstraktionsnivån inspiration och kunskapen för arbetet att forma en emulatormodell åt IOM 200.
- Utvecklat en emulatormodellen i form av en plattform i Linux, som består av en FreeRTOS-simulator med en samling gränssnitt. Gränssnittet har designats av arbetet och är den kod som möjliggör IOM 200 applikationer att exekvera på FreeRTOS-simulatorn.
- Utvecklat en fungerande emulatorprototyp som realiserade ett mindre kodsegment från IOM 200 applikationen.

Emulatormodellen för IOM 200 har potential att generaliseras för andra inbyggda system som använder operativsystemet FreeRTOS. Därför rekommenderas fortsatta arbeten att undersöka möjligheten att generalisera emulatormodellen på så sätt att den kan emulera flera och olika typer av inbyggda system som använder operativsystemet FreeRTOS. Arbetet vill även rekommendera DeLaval att bygga klart emulatorprototypen och sedan generalisera den för resten av deras inbyggda system.

För fortsatta arbeten rekommenderas djupare undersökningar i vinsterna av att implementera en emulator vid utveckling och testning av mjukvaran till ett inbyggt system. Det rekommenderas även att undersöka hur avbrottshanteringen i ett inbyggt system kan realiseras i en emulator.

Källförteckning

1. DeLaval. Om DeLaval [Internet]. [citerad 2021 Mars 29]. Hämtad från: https://www.delaval.com/sv/om-delaval

2. Khorikov V. Unit Testing Principles, Practices, and Patterns. Shelter Island: Manning Publications; 2020. Chapter 2. What is a unit test?

3. Brandl M, Kellner K. Performance Evaluation of Power-Line Communication Systems for LIN-Bus Based Data Transmission. MDPI AG. 2021 Januari 4; p. 10(1):85.

4. DeLaval. Mjölkningsstallar [Internet]. [citerad 2021 April 15]. Hämtad från: https://www.delaval.com/sv/utforska/mjolkning/mjolkningsstallar/

5. DeLaval. Roterande mjölkningsstallar [Internet]. [citerad 2021 April 15]. Hämtad från: https://www.delaval.com/sv/utforska/mjolkning/roterande-mjolkningsstallar/

6. Cypress Semiconductor Corporation. PSoC® 4: 4200_BLE Family Datasheet [Datablad]. 2016 February 22.

7. Zaks R. Från kretsar till system: en introduktion till mikroprocessorer. Stockholm: Pagina Förlag AB; 1982. Kapitel 1: Grundläggande begrepp; s. 1-37.

8. Yiu J. In The Definitive Guide to the ARM Cortex-M0. Newnes; 2011. Kapitel 2, Cortex-M0 Technical Overview; s. 13-24.

9. Cypress Semiconductor. PRoC BLE Architecture Technical Reference Manual [Datablad]. 2017 Juni 6.

10. Bluetooth SIG. About Bluetooth [Internet]. [citerad 2021 April 13]. Hämtad från: https://www.bluetooth.com/about-us/

11. Hirzel T. Arduino tutorial on PWM [Internet]. 2018 [citerad 2021 April 13]. Hämtad från: https://www.arduino.cc/en/Tutorial/Foundations/PWM /

12. Walls C. Embedded RTOS Design. Newnes; 2021.

13. FreeRTOS. About FreeRTOS organisation [Internet]. [citerad 2021 April 14]. Hämtad från: https://www.freertos.org/about-RTOS.html/

14. AlKabary A. Learn Linux Quickly. Packt Publishing; 2020. Your First Keystrokes.

15. Ubuntu. About Ubuntu [Internet]. [citerad 2021 April 15]. Hämtad från: https://ubuntu.com/about/

16. FreeRTOS. Posix/Linux Simulator Demo for FreeRTOS [Internet]. [citerad 2021 April 23]. Hämtad från: https://www.freertos.org/FreeRTOS-simulator-for-Linux.html/

17. Lewine DA. POSIX Programmer's Guide. O'Reilly; 1991. Preface. s. xxiii-1.

18. National Library of the Netherlands. What is emulation? [Internet]. [citerad 2021 Maj 03]. Hämtad från: https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation/

19. Wicaksana A, Tang CM. Virtual Prototyping Platform for Multiprocessor System-on-Chip Hardware/Software Co-design and Co-verification. Computer and Information Science. Springer International Publishing; 2018. s. 93-108.

20. Huang C, Yang K, Chang Y, Wu C, Chen S. A Tiny Development Platform with Virtualized Peripherals for Education of Embedded Software Design. Advanced Materials Research. 2013; 748.

21. Heunhe Han A, Hwang YS, An YH, Lee SJ, Chung KS. Virtual ARM Platform for Embedded System Developers. International Conference on Audio, Language and Image Processing. 2008; Shanghai. s. 586-592.

22. Hadipurnawan S, Budiono W, Jin B. K, Jeong B. L, Young S. H. VDEES: A Virtual Development Environment for Embedded Software Using Open Source Software. IEEE Transactions on Consumer Electronics. 2009; 55(2).

TRITA TRITA-CBH-GRU-2021:052