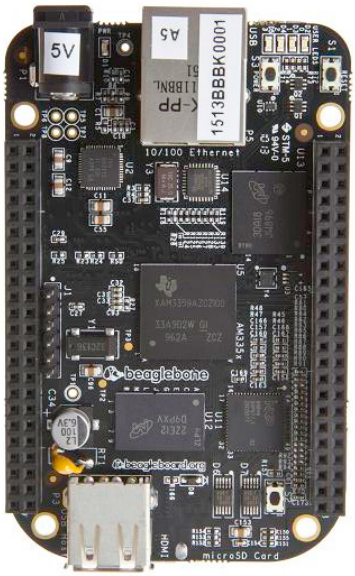
CHALMERS




# Dialux - ett program för processdiagnostik i en embedded Linux miljö

## Dialux - a program for process diagnostics in an embedded Linux environment

Examensarbete inom Mekatronik

Emil Eide

Robin Linder Saied

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

EXAMENSARBETE

# Dialux

**Ett program för processdiagnostik i en embedded Linux miljö**

Emil Eide
Robin Linder Saied

**Institutionen för Data- och Informationsteknik**
**CHALMERS TEKNISKA HÖGSKOLA**
**GÖTEBORGS UNIVERSITET**

**Göteborg 2019**

**Dialux**
Ett program för processdiagnostik i en embedded Linux miljö
EMIL EIDE
ROBIN LINDER SAIED



Examinator: Peter Lundin

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
412 96 Göteborg
Telefon: 031-772 1000



Institutionen för Data- och Informationsteknik
Göteborg 2019

# Sammanfattning

Embedded Linux används allt mer inom olika produkter och system. Trots detta finns det en brist på diagnostiseringsprogram för denna typ av system. Målet med detta projekt är att påbörja utvecklingen av ett enkelt diagnostiseringsprogram för en process som körs på embedded Linux. Det inbyggda system som används är en Beaglebone Black. Arbetet utförs på Diadrom Systems AB som är ett specialistföretag med fokus på diagnostik av autotech. Arbetet har avgränsats till att övervaka en process och presentera körningsvärden för CPU- och RAM-användning i ett användarvänligt format. Frågor som behandlas är hur nödvändiga parametrar ska hämtas, när dom ska hämtas, hur ofta läsning ska ske, samt hur datan ska presenteras. Ett diagnostiseringsprogram kallat Dialux skapades med C/C++-programmering. Detta gjordes genom utveckling av tre olika iterationer. Ett testprogram skapades också för att validera Dialux funktion. Resultatet blev att Dialux hämtar körningsvärden från det virtuella filsystemet proc. Det använder sig av continuous built in test med ett tidsintervall på två sekunder. Körningsdatan presenteras i ett Excel dokument.

# Abstract

Embedded Linux is increasingly used in various products and systems. Despite this, there is a great lack of diagnostic programs for this type of system. The goal of this project is to start the development of a simple diagnostics program for a process that runs on embedded Linux. The work is carried out at Diadrom Systems AB, which is a specialist company with focus on autotech diagnostics. The work has been limited to monitoring a single process and presenting run values for CPU and RAM usage in a user-friendly format. The main questions that are dealt with are how to retrieve the necessary parameters, when should they be retrieved, how often reading should be done and how the data should be presented. A diagnostics program called Dialux was created with C / C ++ programming. This was done by developing three different iterations. A test program was also created to validate Dialux's function. As a result, Dialux retrieves run values from the proc virtual file system. The most suitable built in test is the continuous built in test with the chosen time interval of two seconds. The data is presented in an Excel document.

# Förord

Examensarbetet utförs på Chalmers tekniska högskola och omfattar 15 högskolepoäng. Det görs för programmet Mekatronik som är 180 högskolepoäng tillsammans med företaget Diadrom Systems AB.

Vi vill tacka Henrik Fagrell, styrelseordförande på Diadrom, som varit vår handledare på företaget samt alla kollegor på Diadrom. Vi vill även tacka Roger Johansson som varit vår handledare på Chalmers samt Peter Lundin som är vår examinator.

# Innehåll

# 1

# Inledning

## 1.1 Bakgrund

Embedded Linux används allt mer inom olika produkter och system[1]. Trots detta finns det en brist på diagnostiseringsprogram för processer som kör på denna typ av inbyggd mjukvara. Med detta i åtanke är målet med detta projekt att påbörja utvecklingen av ett program som uppfyller detta behov. Projektet har utförts under vårterminen 2019 på Diadrom Systems AB. Diadrom är ett specialistföretag med fokus på diagnostik av autotech [2]. Företagets inriktning är främst riktat mot fordonsindustrin men även mot företag inom försvar, publik transport och säkerhet. Diadrom arbetar med mjukvara över hela produktens livscykel vilket innefattar till exempel uppladdning av mjukvara, konfigurering och diagnostik. Det är inom diagnostisering detta projekt är utformat.

## 1.2 Syfte

Förväntan på projektet är att ett enkelt diagnostiseringsprogram av en process som körs på embedded Linux ska utvecklas. Programmet ska övervaka en process och presentera körningsvärden preciserat till CPU- och RAM-användning i ett användarvänligt format.

## 1.3 Avgränsningar

Arbetet är begränsat till mjukvaruutveckling då hårdvaran består av en Beaglebone Black [3]. Debian är förinstallerat på enheten och således avgränsas programutvecklingen till ett diagnostiseringsprogram för en process som kör på detta operativsystem [4]. Diagnostiseringen begränsas till de uppsatta målen eftersom omfattningen annars kan bli för stor.

## 1.4 Frågeställning

Hur ska ett diagnostiseringsprogram utformas för att diagnostisera en process i embedded Linux?
Huvudfrågan har delats upp i följande delfrågor:

- Hur ska nödvändiga parametrar för att diagnostisera en process i en embedded Linux miljö hämtas?
- I vilket skede av processen ska läsning av körningsdata ske?
- Hur ofta ska läsning av körningsdata ske?
- Hur ska insamlad data presenteras för en bra översikt av avvikelser mellan flera körningar?

# 2
# Teoretisk bakgrund

I detta avsnitt presenteras information om ingående mjuk- och hårdvara i projektet. Relevant information presenteras för att ge förståelse för projektets syfte, arbetsgång och resultat.

## 2.1 Embedded Linux

Linux är en operativsystemskärna som ursprungligen skrevs av Linus Thorvalds[5]. I linuxkärnan finns de olika kärnsystem som behövs för att kunna köra ett system som är baserat på Linux. Mjukvara som körs på Linux använder sig av specifika funktioner hos linuxkärnan som till exempel virtuellt minne, filer och uppgifter. Ett linuxsystem eller en distribution är byggd på linuxkärnan med specifik mjukvara.

Linuxkärnan är kopplad till hårdvaran och syftet med kärnan är att hantera hårdvaran samtidigt som den förser användaren med abstraktioner på högnivå. Därmed hanterar linuxkärnan I/O-portar, minneshantering, filsystem och nätverksprotokoll samtidigt som den ger möjlighet för användaren att få åtkomst till detta med mjukvara på högnivå. Linuxkärnan är alltså länken mellan hårdvaran och applikationer som körs på systemet. Ett exempel på detta är GNU/Linux som använder sig av GNU mjukvara utvecklat av Rickard Stallman och The Free Software Foundation. Distributionen som används i detta projekt är Debian GNU/Linux som är byggt på operativsystemskärnan Linux samt GNU-projektet.

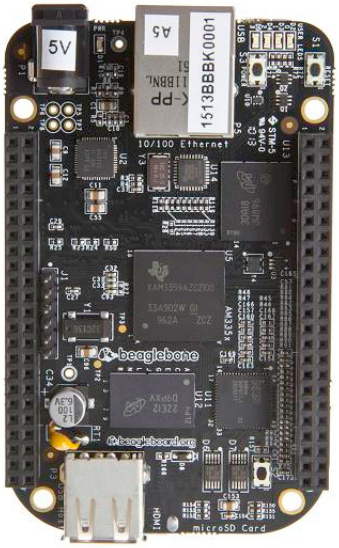
Embedded Linux bygger på linuxkärnan, men är avsedd för inbyggda system. Inbyggda system utför ofta specifika funktioner som är mindre omfattande än till exempel en persondator. Detta medför att det krävs ett operativsystem som är effektivt utformat för det inbyggda systemets funktion. För detta finns det flertalet linuxdistributioner som kan användas. Men eftersom linuxkärnan är open-source så kan även egna skräddarsydda distributioner tas fram utefter applikationens krav.

## 2.2 Beaglebone Black

För implementering av diagnostiseringsprogrammet på ett embedded Linux system används en Beaglebone Black. Denna enkortsdator är utrustad med:

- 512 MB RAM
- 4 GB flashminne
- AM335x 1GHz ARM Cortex-A8 processor
- Micro-USB-anslutning för ström och dataöverföring
- Ethernet-anslutning

Den är kompatibel med ett flertal linuxbaserade operativsystem som android, ubuntu och debian[3]. För detta projekt används det sistnämnda. En bild på enheten kan ses i figur 2.1 nedan.




**Figur 2.1:** Beaglebone Black. Från [3]. Återgiven enligt creative commons attribution-share alike 3.0 license.

## 2.3 Parametrar som ska diagnostiseras

Diagnostik av en process i embedded Linux innebär att övervaka processens beteende och se efter avvikelser. Avvikelser hos processen kan uppstå i flera av processens läsbara parametrar. Respektive parameter säger olika mycket om hur processen uppför sig. Ett inbyggt system har ofta relativt nedskalad hårdvara och därför är det viktigt att den används på ett effektivt sätt[5]. Med detta i åtanke har parametrarna som ska övervakas avgränsats till processens RAM- och CPU-användning. Även systemets totala RAM- och CPU-användning övervakas då hög total förbrukning kan inverka på processen.

## 2.4 CPU-användning

För att beräkna hur stor del av processorns kapacitet som används av processen, behöver två mätningar göras över en bestämd tidsperiod[6]. Tiden som processorn är upptagen med att utföra en viss process jämförs med tiden den går på tomgång under det bestämda tidsintervallet. Som exempel kan ett tidsintervall vara 10 sekunder långt. Om processorn under detta intervall är upptagen i 4 sekunder, innebär det att 40 procent av processorns kapacitet använts. Det procentuella värdet är ett medelvärde över tidsintervallet, denna tid kan minskas för att ge ett närmare momentant värde. Detta gäller vid användning av en enda processor och är det som används i detta projekt. Vid användning av flera processorer måste värdet delas med antalet processorer för att få fram ett korrekt värde.

## 2.5 RAM-användning

Det finns flera olika tillvägagångsätt för att mäta hur stor mängd minne en process använder. Två vanliga mätvärden är VSS (virtual set size) och RSS (resident set size)[7]. Båda mätvärdena går att hämta ur det virtuella filsystemet proc som skapas av linuxkärnan [8]. VSS inkluderar allt minne som processen har åtkomst till. Det innefattar även utbytt minne, allokerat men ej använt minne samt minne från delade bibliotek. Eftersom en stor del av det virtuella minnet aldrig används av det fysiska minnet så är VSS av mindre intresse vid diagnostik.

RSS visar summan av minnet som är länkat till de fysiska sidorna av minnet. Detta inkluderar minne på stacken och heap:en. Även minne från delade bibliotek så länge de sidorna faktiskt används i minnet. Däremot inkluderas ej minne som har bytts ut och minne som allokerats men ej används. Detta gör att RSS visar värden mer lika den faktiska minnesanvändningen jämfört med VSS.

## 2.6 C/C++

Programmeringsspråket C utvecklades i början av 1970-talet och är ett högnivåspråk som kan användas för programmering av hårdvara på en lägre nivå[9]. Även om C har flera olika användningsområden så var det ursprungligen designat för att programmera systemapplikationer. Därmed är det ett flexibelt och kraftfullt språk med effektiv exekvering av program. Detta gör att C passar bra för programmering av inbyggda system. C++ härstammar och är kompatibelt med C samtidigt som det stödjer objekt-orienterad programmering[10]. Det är reformerat med utbyggt typ-system vilket gör C++ säkrare och med ett kompileringssystem som enklare detekterar fel i koden. Detta har medfört att både C och C++ används i detta projekt.

## 2.7 CSV

För att kunna överföra den insamlade datan till ett Excel dokument så är ett sätt att först lagra datan i en textfil. En CSV (Comma-separated Values) fil är ett textfilsformat som används i stor utsträckning för att lagra data i en tabell[11]. Denna tabell är uppbyggd som en matris där översta raden innehåller parameternamnen, denna rad kallas CSV-filens huvud. Parametrarna och parametrarnas värden separeras av en avdelare, ofta ett kommatecken eller ett semikolon. En fördel med CSV format är att det är nästintill universalt och kan exporteras till flera program som till exempel Microsoft Excel.

# 3
# Metod

För att uppfylla uppsatta mål kommer arbetet främst att ske på Diadroms kontor där stor tillgång till information och kompetens finns. Arbetet ska inledas med relevant datainsamling för en ökad förståelse om embedded Linux och hårdvaran i en Beaglebone Black. Detta ska delvis ske på diverse hemsidor som till exempel beagleboard.com samt genom kommunikation med utvecklare på Diadrom. Därefter ska information insamlas för att skapa en teoretisk bild av hur programmet ska vara uppbyggt för att kunna uppnå målen. Detta ska främst ske via workshops och kommunikation med utvecklare på Diadrom.

Med hjälp av anskaffad information om hur programmet ska konstrueras ska en prototyp tas fram. Prototypen ska tas fram stegvis utefter varje mål och testas för varje delmål. Först måste det bestämmas hur information till parametrarna kan hämtas för att sedan avgöra vilket sätt som är mest lämpligt. Sedan måste det bestämmas när diagnostiseringsprogrammet ska hämta data och hur ofta. Till sist måste ett sätt att presentera data väljas. Detta ska kunna ge användaren en bra översikt av hur körningarna ser ut och underlätta för att hitta avvikelser.

För att simulera de fel som programmet ska kunna upptäcka i en process, kommer ett testprogram skrivas vars syfte är att ge avvikande körningsvärden. Båda programmen kommer att skrivas med programspråken C++ och C i Visual Studio 2017. Testningen ska ske i samråd med Diadrom och på Beaglebone:n. När testningen av prototypen är gjord ska eventuella fel åtgärdas för att sedan skapa den fullständiga produkten.

# 4
# Datainsamling

Under arbetets gång har information hämtats för att bygga en förståelse för hur embedded Linux är uppbyggt och hur kod ska implementeras för att skapa Dialux. Information om linuxkärnan och dess funktioner gjordes via Chalmers biblioteket där det fanns tillgång till flera böcker och vetenskapliga artiklar som behandlar Linux. Eftersom Linux är open-source finns det även mycket information och manualer tillgängligt på Linux officiella hemsida. Information om Beaglebone Black erhålldes på tillverkarens hemsida samt genom ett informationsblad tillhörande enheten [12].

För C/C++ utvecklingen hämtades information delvis från boken C från början [13], som tidigare används i kursen programutveckling på Chalmers. Denna bok behandlar grundläggande C-programmering. Ytterligare information om C/C++-programmering hämtades på flera hemsidor och forum. Exempel på detta är Stackoverflow där det finns information och diskussioner om flertalet funktioner som är relevanta för detta arbete.

# 5

# Systemkonstruktion

För att genomföra utvecklingen av Dialux har arbetet delats upp i flera delmål. Detta eftersom det finns flera olika sätt att konstruera programmet och att flera parametrar måste övervägas innan beslut tas. Arbetsgången har följt ordningsföljden av de fyra frågeställningar som ställts. Svar på respektive frågeställning har framtagits enligt de metoder som presenteras i metod-avsnittet ovan.

## 5.1 Hämtning av parametervärden

Hur parametervärden hämtas är en central del av ett diagnostiseringsprogram och kan variera mellan olika system. Det beror även på vilka parametrar som ska användas och om det är inriktat på hårdvara eller systeminformation. Eftersom detta program behandlar embedded Linux så finns det flera linuxspecifika sätt att avläsa information. Linuxkärnan sköter kommunikationen mellan processer, hårdvaran och systemet genom systemanrop[10]. Detta möjliggör att kärnan kan lagra information om samtliga processer i kataloger. Dialux ska övervaka en enskild process och behöver således värden för enskilda processer. Detta gör att linuxkärnans kataloger blir användbara.

Filsystemet som möjliggör kommunikation mellan applikationer och linuxkärnan kallas proc[14]. Detta filsystem skapades för att presentera information om processer som körs och det är även i detta syfte filsystemet används i detta projekt. Filsystemet är uppbyggt av flera olika kataloger med information. Den nödvändiga informationen för att beräkna CPU- och RAM-användning finns i dessa kataloger under proc/stat respektive proc/meminfo. För att få värden för den enskilda processen så används dess PID (process ID). Ett PID skapas när processen initieras och är unikt för just den processen. De specifika värdena som hämtas och hur de beräknas beskrivs i sektionen beräkningar. Eftersom informationen i proc är omfattande kan även total CPU- och RAM-användning hämtas här. Att använda proc filsystemet är ett smidigt och enkelt sätt att hämta processinformation. Eftersom linuxkärnan alltid lagrar detta så skapar det heller inte några extra påfrestande anrop.

## 5.2 När programmet ska exekvera

När frågan om hur körningsvärden för processen skulle hämtas var besvarad. Fortsatte arbetet med att bestämma när och hur ofta körningsvärden skulle hämtas. Efter konsultation med personal på Diadrom blev slutsatsen att de nedanstående fyra alternativen var att utgå från [15].

- PBIT - Power up built in test
  - Diagnostisering sker vid processens start för att se efter avvikelser. Skulle en avvikelse upptäckas så skulle det vara möjligt att avbryta processen innan den påbörjas för att förhindra att fel inträffar.

- CBIT - Continuous built in test
  - Diagnostiseringsprogrammet initieras vid uppstart av processen och samlar körningsdata kontinuerligt i bakgrunden.

- IBIT - Initiated built in test
  - Diagnostiseringsprogrammet initieras vid anrop av användaren under körning. Anropas vanligen i samband med att ett fel upptäcks vid CBIT.

- PDIT - Power down built in test
  - Diagnostiseringen påbörjas precis innan processen avslutas, stör därmed inte under själva körningen. Eventuella fel presenteras efter processen avslutats.

Två av de fyra alternativen ovan testades, dock inte PBIT och IBIT. Eftersom de inte ansågs lämpliga för syftet, samtidigt som avgränsningar gjordes för att kunna slutföra projektet inom given tidsram. PBIT är bättre lämpat för processer där avvikelser kan innebära en säkerhetsrisk [15]. Alltså att processen då måste klara diagnostiken innan den exekveras. IBIT anropas av användaren och är således inte passande för detta projekt eftersom diagnostiken ska kunna utföras även om ett fel inte har uppkommit.

Vid diskussion i planeringsstadiet framkom PDIT alternativet. Detta ansågs vara ett möjligt alternativ att implementera eftersom användaren då skulle få alla parametrar presenterade efter körningen avslutats samtidigt som diagnostiken inte stör processen under körningens gång. Detta alternativ visade sig dock svårt att genomföra eftersom en diagnostisering som körs när processen avslutats inte får tillgång till parametervärden löpande från processen utan endast ett medelvärde för hela körningen. Detta på grund av hur proc är uppbyggt, där endast momentana värden går att hämta. Ett medelvärde för CPU- och RAM-användning ansågs inte passande för detta projekt eftersom programmet ska kunna visa avvikelser och ge möjlighet till analys av hela körningen.

Till slut valdes CBIT eftersom detta alternativ ger möjlighet till insamling av parametervärden kontinuerligt under hela körningen. Därmed samlas mer användbar information in och underlättar för analys där grafer och diagram kan visualisera körningen. Detta medför dock att diagnostiseringsprogrammet körs i bakgrunden under körningen och blir mer krävande. Men efter att tester genomförts bedömdes ändå detta vara det mest lämpliga alternativet eftersom det uppfyller projektets syfte om att presentera körningsvärden för processen.

## 5.3 Beräkningar

Med de insamlade värdena från proc kunde CPU- och RAM-användning beräknas. Nedan beskrivs vilka värden som hämtas samt hur beräkningarna genomförts.

### 5.3.1 CPU-användning

I katalogerna proc/[PID]/stat och proc/stat hämtas parametrarna listade nedan som används för att beräkna processens och den totala CPU-användningen[8]. Samtliga är i enheten jiffies/clock ticks, vilket är en tidsreferens i ett linuxsystem[16]. Längden av en jiffie varierar beroende på systemarkitektur. Nedan följer de parametrar som hämtas, samt en kort beskrivning av varje.

- utime - tid som den övervakade processen spenderat i användarläge.
- stime - tid som den övervakade processen spenderat i kärnan.
- starttime - tid som passerat sen processen startades
- user - tid som samtliga processer spenderat i användarläge.
- nice - tid som samtliga processer spenderat i användarläge med låg prioritet.
- system - tid som samtliga processer spenderat i systemläge.
- idle - tid som samtliga processer spenderat i tomgångsläge.

- iowait - tid som samtliga processer spenderat på att vänta på I/O.
- irq - tid som samtliga processer spenderat på underhåll av avbrott.
- softirq - tid som samtliga processer spenderat på underhåll av mjuvaruavbrott.

Samtliga parametrar läses in två gånger med ett bestämt tidsintervall mellan läsningarna. Genomsnittliga CPU-användningen under intervallet kan sedan beräknas enligt ekvation 5.1 nedan[17].

$$\frac{(utime + stime) - (utime_{old} + stime_{old})}{starttime - starttime_{old}} \times 100 \tag{5.1}$$

Ekvationen ger ett procentuellt värde för hur stor del av intervallet som processorn varit upptagen. Vilket är CPU-användningen för processen.

Beräkning av den totala CPU-användningen görs på liknande sätt, dock med andra parametrar vilket kan ses i ekvation 5.2 nedan:

$$\frac{WorkTime - WorkTime_{old}}{TotalTime - TotalTime_{old}} \times 100 \tag{5.2}$$

där:

$$WorkTime = user + nice + system$$

$$TotalTime = user + nice + system + idle + iowait + irq + softirq$$

$WorkTime$ är den tid som processorn varit upptagen med samtliga processer som kör på systemet, uttryckt i jiffies. $TotalTime$ är summan av tiden som passerat då processorn både varit upptagen och inaktiv uttryckt i jiffies. Differensen av tiden då processorn varit upptagen mätt vid två tillfällen, ställs i förhållande till differensen av den totala tid som passerat under tidsintervallet. Detta ger den genomsnittliga CPU-användningen över tidsintervallet.

### 5.3.2 RAM-användning

Värdet som hämtas för att beräkna RAM-användning för processen är RSS som innehåller antalet pages i det fysiska minnet. Detta värde hämtas i katalogen proc/[PID]/stat. Storleken på en page kan variera och beror på processorn[18]. processorns minneshanterare omvandlar virtuella adresser till fysiska adresser. De flesta 32-bitars processorer har en minneshanterare som stödjer 4 kB per page. Detta gäller även för ARM Cortex-A8 som är processorn som används i detta projekt[19]. Parametern RAM-användning ansågs vara mest lämpligt att presentera i kB så därmed behövdes en konvertering göras från pages till kB. Detta gjordes enligt nedanstående ekvation där antalet pages multipliceras med 4096 för att få antalet i bytes. Detta delas sedan med 1000 för att få fram värdet i kB.

$$RAM = \frac{rss \times 4096}{1000} \tag{5.3}$$

För att beräkna den totala RAM-användningen används MemTotal samt MemFree. Dessa hämtas från katalogen /proc/meminfo[8].

- MemTotal - Totala användbara RAM. Det fysiska RAM-minnet bortsett från några reserverade bitar samt linuxkärnans binärkod.

- MemFree - Totala fysiska RAM-minnet som inte används av systemet.

Både MemTotal och MemFree erhålls i kB och behöver därmed inte konverteras eftersom de redan är i önskat format. För detta projekt sökes den totala använda RAM-användningen för att kunna jämföra detta med hur mycket den enskilda processen förbrukar. Totala fysiska RAM som används fås genom ekvationen nedan där totala användbara RAM subtraheras med totala lediga RAM.

$$TotRam = (MemTotal - MemFree) \tag{5.4}$$

## 5.4 Presentation av data

När ett system för insamling av data hade upprättats samt beräkningar för parametervärden utförts behövdes ett enkelt sätt att lagra och presentera data. Ett alternativ för att lagra data var XML (Extensible Markup Language). XML är ett märkspråk som kan användas för att skapa ett dataregister där det är en skillnad mellan data och presentation[20]. Här används taggar och leder till att både människa och maskin kan urskilja vad som är data, till exempel ett procentuellt värde på CPU-användning, samt kategorin CPU-användning. Efter att ha implementerat XML var den samlade bedömningen att detta inte var det mest lämpliga sättet att presentera utefter projektets målsättning. Detta eftersom projektet var avgränsat till två parametrar och att det då inte krävs ett sådant dataregister.

Ett annat alternativ som också implementerades var JSON (JavaScript Object Notation). JSON är ett textbaserat dataformat som utrycker objekt som en lista av egenskaper som är läsligt för människor[21]. Med parametrarna som används i detta projekt skulle till exempel CPU-användning vara ett objekt och det procentuella värdet vara egenskapen. JSON kan vara ett smidigt sätt att lagra data och är även språkoberoende. Efter att ha testat detta alternativ var bedömningen likt XML att det inte krävs ett sådant dataregister för den mängden av data som används. Eftersom det även behövs parsing leder detta till mer arbete och komplexitet.

Det mest lämpliga sättet att presentera datan bedömdes vara i form av en CSV fil. Detta eftersom en sådan textfil nästintill är universal och kan överföras till andra program där datan kan beskrivas visuellt. För detta projekt överförs textfilen till ett Excel dokument där grafer kan skapas och analyseras. Med CSV formatet skapas mer frihet för användaren att visualisera datan samtidigt som det är mindre komplext att hantera. Eftersom mängden data som samlas in är begränsad så går det även att hantera datamängden utan att behöva skapa ett dataregister likt XML och JSON.

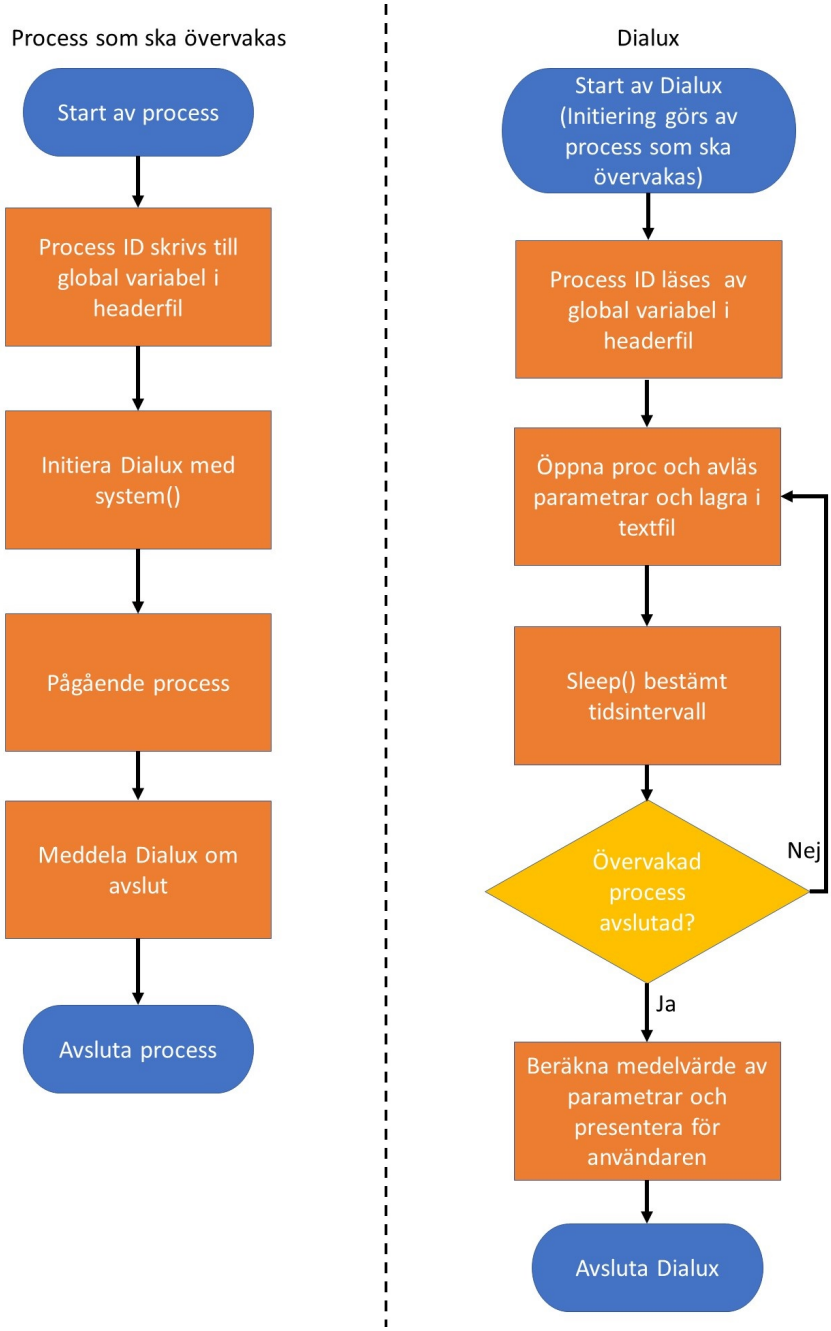
Till CSV filen läses tidstämplade värden från parametrarna CPU- och RAM-användning. Tidstämpeln innefattar datum och tid. För parametrarna läses information om processvärde och den totala användningen. Detta gör att både CPU- och RAM-användning kan jämföras med respektives totala användning. Detta ger en mer tydlig beskrivning av processens användning och kan beskrivas visuellt i olika grafer. Den graf som valts i Excel är linjediagram eftersom detta passar bra vid tidsberoende värden och även ger en visualisering av både processens och totala förbrukningen.

## 5.5 Programarkitektur

Under projektets gång har totalt tre olika iterationer av programmet tagits fram. Samtliga bygger på liknande principer, men de skiljer sig åt i hur de opererar. För att kunna testa att övervakningen fungerar skrevs även ett program med syftet att skapa avvikelser som Dialux kan upptäcka och övervaka. Detta program utförde beräkningar för att belasta processorn, samt minnesallokeringar för att ta upp stor plats i systemets RAM. Nedan följer respektive iterations struktur samt förklaring till hur de alla fungerar. Processen som övervakas illustreras inte i större detalj i respektive iterations flödesschema.

### 5.5.1 Iteration 1

Första iterationen av programmet kom aldrig till att bli ett fullt fungerande program utifrån syftet. Dess struktur framtogs i ett tidigt skede av datainsamlingen, då kunskapen om embedded Linux inte var fullt utvecklad inom projektgruppen. I Figur 5.1 nedan illustreras dess tilltänkta struktur i ett flödesschema som togs fram i början av projektet.




**Figur 5.1:** Flödesschema för iteration 1

Bilden visar två olika program där det vänstra programmet är den process som ska övervakas, det högra är Dialux. En körning påbörjas med att processen skriver sitt process-ID till en headerfil som en global variabel. Därefter initieras Dialux med *system()* kommandot. Dialux läser in processens PID från den gemensamma hearderfilen. Med process ID:t hämtas sedan nödvändiga parametrar för att kunna beräkna CPU-användningen för processen, dessa skrivs till en textfil.

Dialux väntar sedan ett bestämt tidsintervall, om processen fortfarande kör görs en ny läsning från proc och textfilen utökas med fler värden. När processen sedan avslutas beräknar Dialux ett medelvärde för processens CPU-användning under körningen och presenterar det i text för användaren i terminalen.

Denna iteration hade flera problem och funkade därmed inte helt enligt flödesschemat i figur 5.1 som togs fram innan programkoden skrevs. Flera läsningar under en körning var ej möjligt på grund av hur *system()* kommandot fungerar. *System()* låter ej processerna köra parallellt, istället väntar processen som kallar *system()* på att den anropade processen ska bli klar[22].

Att placera processens PID i en headerfil gjorde att tillgång till processens källkod krävdes. Effektivare sätt att hämta processens PID användes därför i nästkommande iterationer. Skrivning och läsning till en .txt fil utan någon separerare eller struktur visade sig också vara problematiskt, främst vid läsning av data. Med Iteration 1:s brister och fördelar i åtanke fortsatte utvecklingen genom påbörjandet av iteration 2.

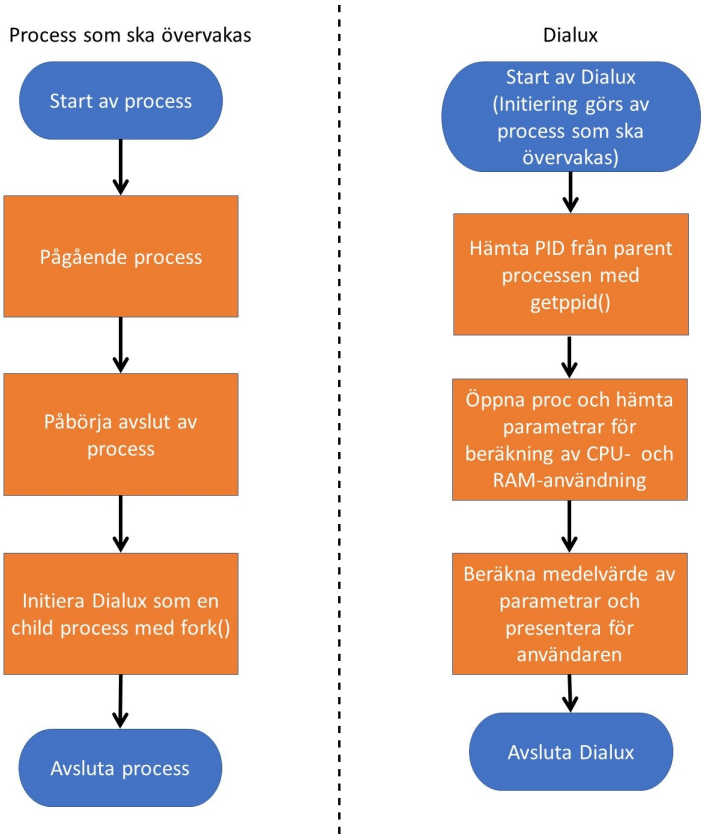
### 5.5.2 Iteration 2

Likt iteration 1 initieras Dialux av processen som ska övervakas, detta görs däremot i slutet av processen istället för i början, vilket kan ses i figur 5.2 på nästkommande sida. PDIT konceptet testades därmed i denna iteration som ett försök att konstruera programmet så det påverkar processen minimalt under körning.

Dialux öppnas genom att processen använder kommandot *fork()* för att initiera en child process. Processen som gör anropet kallas då för en parent process. I child processen används sedan *exec()* för att initiera Dialux. Detta ersätter den tidigare initieringen i iteration 1 som gjordes med *system()*. Förvisso använder *system()* både *fork()* och *exec()*, dock även några kommandon ytterligare vilket gör *system()* enkelt men ineffektivt[22].

När Dialux initierats hämtas processens PID med *getppid()* vilket returnerar parent processens process ID. Med parent processens PID hämtar Dialux parametrar för beräkning av CPU- och RAM-användning från proc. Ett medelvärde för hela körningen för CPU- och RAM-användning beräknas och presenteras för användaren. I iteration 2 testades flera presentationsformat. XML, JSON och CSV implementerades, valet föll på CSV i slutändan för minskad komplexitet.

Iteration 2 var en förbättring av iteration 1 då den åtgärdade en del av den tidigares brister. Däremot kvarstod problemet med att det krävdes tillgång till processens källkod. Medelvärden av CPU-användning och allokerat RAM för en hel körning ansågs dessutom inte uppfylla syftet. Därför fortsatte utvecklingen med en ytterligare iteration.



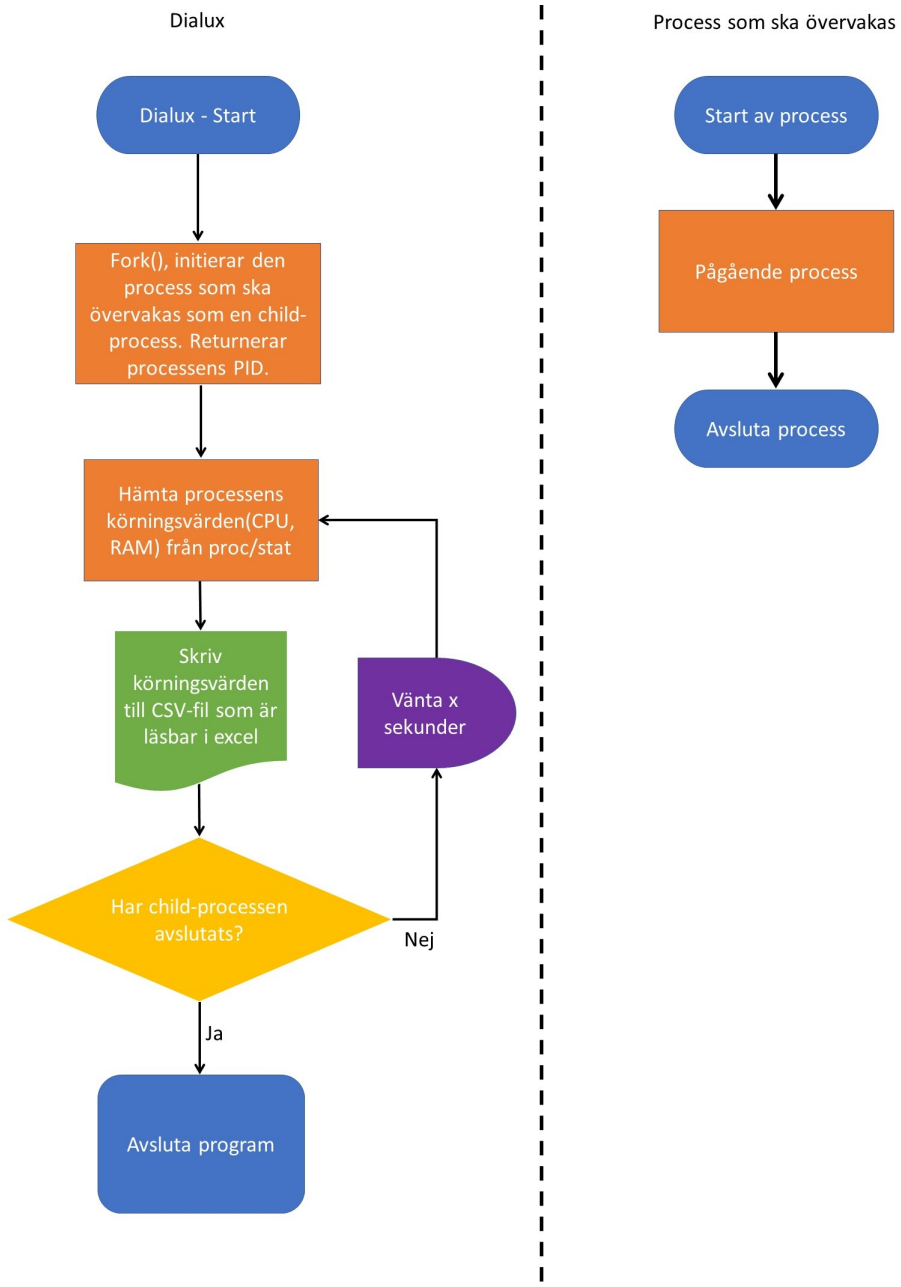

**Figur 5.2:** Flödesschema för iteration 2

### 5.5.3 Iteration 3

Iteration 3 är den slutgiltiga versionen av programmet som tagits fram under projektet. Den uppfyller projektets syfte och besvarar samtliga frågeställningar. Största ändringen i programmets struktur till skillnad från tidigare iterationer, är att Dialux nu initierar den process som ska övervakas. Detta görs för att undvika behovet av redigering i källkoden för processen som ska övervakas. På så sätt möjliggörs diagnostik av processer där tillgång till källkod saknas. I figur 5.3 nedan illustreras den förändrade strukturen i ett flödesschema för tydlighet och enkel jämförelse med tidigare iterationer.

När Dialux startats initierar den processen som ska övervakas som en child process med hjälp av kommandot *fork()* samt *exec()*. Med *fork()* kommandot returneras child processens PID, vilket är processen som ska övervakas, denna process startas med *exec()*. Programmen kör nu parallellt med varandra fram till att child processen avslutas. Under tiden som child processen kör, hämtar Dialux parametrar från proc

för beräkning av CPU- och RAM-användning för processen. Även totala CPU- och RAM-användningen för enheten beräknas. Parametrarna ges en tidsstämpel och skrivs därefter till en CSV-fil. Läsning från proc görs flera gånger under en körning med ett bestämt tidsintervall mellan läsningarna. Vid de tester som gjorts användes ett tidsintervall på två sekunder. Dialux kollar kontinuerligt om child processen avslutats, vid avslut går Dialux till programslut.




**Figur 5.3:** Flödesschema för iteration 3

# 6
# Resultat

## 6.1 Besvarande av frågeställningar

- Hur ska nödvändiga parametrar för att diagnostisera en process i en embedded Linux miljö hämtas?

  Värden för de valda parametrarna kan hämtas i det virtuella filsystemet proc som erhålls av linuxkärnan. Värdena för en enskild process kan hämtas genom att använda processens PID nummer i proc. Med hjälp av dessa värden kan sedan parametrarna beräknas.

- I vilket skede av processen ska läsning av körningsdata ske?

  Det mest lämpliga körningssättet för detta projekt är CBIT. Detta eftersom parametervärden då samlas in kontinuerligt under hela körningen och kan på så vis visualisera hela körningen i form av grafer.

- Hur ofta ska läsning av körningsdata ske för en optimal användarupplevelse?

  Intervallet för inläsning av data är två sekunder. Detta för att få tillräckligt med data till parametrarna även under en körning på ett fåtal sekunder. Dock är detta intervall justerbart och bör ställas efter användningsområde.

- Hur ska insamlad data presenteras för en bra översikt av avvikelser mellan flera körningar?

  Datan ska samlas in i en CSV-fil som är nästintill universal. Datan kan sedan exporteras till Excel där den kan visualiseras i form av grafer. Datan är tidstämplad och skrivs till CSV-filen. Diagrammet som valts för både CPU- och RAM-användning är linjediagram.

## 6.2 Excel dokument

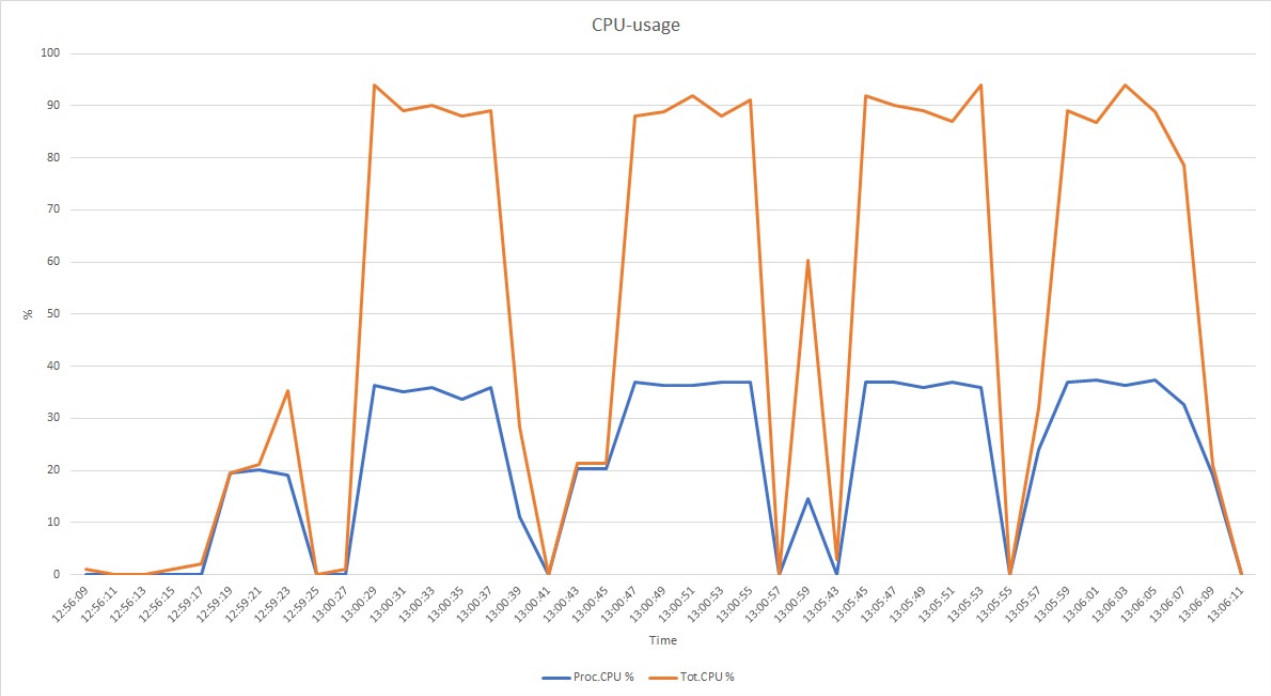
I figur 6.1 nedan visas ett exempel på utskrift från två körningar. Tidstämpeln visas till vänster utefter datum och tid. Parametrarna som skrivs är enligt följande ordning: processens CPU-användning, total CPU-användning, processens RAM-användning, total RAM-användning. För varje ny körning så fylls dokumentet på med mer data. För en bättre översikt kan körningsdatan formateras till tabeller i Excel likt bilden nedan. Detta har även kontrollerats och jämförts med värden som kan ses i Linux kommandot TOP som är ett program för processövervakning i realtid [23].

| Date | Time | Proc.CPU % | Tot.CPU % | Proc.RAM kB | Tot.RAM kB |
|---|---|---|---|---|---|
| 15/5/2019 | 10:42:33 | 0 | 2 | 1945 | 152244 |
| 15/5/2019 | 10:42:35 | 34.6535 | 91.0891 | 1945 | 152268 |
| 15/5/2019 | 10:42:37 | 33 | 93 | 1945 | 152268 |
| 15/5/2019 | 10:42:39 | 34 | 94 | 1945 | 152268 |
| 15/5/2019 | 10:42:41 | 34 | 92 | 1945 | 152244 |
| 15/5/2019 | 10:42:43 | 33 | 87 | 1945 | 152244 |
| 15/5/2019 | 10:42:45 | 34 | 90 | 1945 | 152236 |
| 15/5/2019 | 10:42:47 | 76.7677 | 95.9596 | 2052 | 152236 |
| 15/5/2019 | 10:42:49 | 100 | 100 | 2052 | 152252 |
| 15/5/2019 | 10:42:51 | 34 | 93 | 2052 | 152252 |
| 15/5/2019 | 10:42:53 | 34 | 88 | 2052 | 152252 |
| 15/5/2019 | 10:42:55 | 34.3434 | 90.9091 | 2052 | 152252 |
| 15/5/2019 | 10:42:57 | 22.2222 | 56.5657 | 2052 | 152244 |
| 15/5/2019 | 10:42:59 | 0 | 0 | 0 | 152244 |
| | | | | | |
| 15/5/2019 | 10:48:23 | 0 | 1.02041 | 1908 | 152360 |
| 15/5/2019 | 10:48:25 | 33 | 91 | 1908 | 152508 |
| 15/5/2019 | 10:48:27 | 34 | 91 | 1908 | 152484 |
| 15/5/2019 | 10:48:29 | 33 | 95 | 1908 | 152484 |
| 15/5/2019 | 10:48:31 | 34.6939 | 85.7143 | 1908 | 152476 |
| 15/5/2019 | 10:48:33 | 34 | 92 | 1908 | 152476 |
| 15/5/2019 | 10:48:35 | 35 | 96 | 1908 | 152476 |
| 15/5/2019 | 10:48:37 | 34.3137 | 90.1961 | 1908 | 152476 |
| 15/5/2019 | 10:48:39 | 35 | 92 | 1908 | 152492 |
| 15/5/2019 | 10:48:41 | 33.6634 | 91.0891 | 1908 | 152492 |
| 15/5/2019 | 10:48:43 | 33.6634 | 89.1089 | 1908 | 152484 |
| 15/5/2019 | 10:48:45 | 10.101 | 28.2828 | 1908 | 152484 |
| 15/5/2019 | 10:48:47 | 0 | 7.14286 | 0 | 152484 |

**Figur 6.1:** Exempel på Excel dokument där datan formaterats till en tabell

## 6.3 Linjediagram

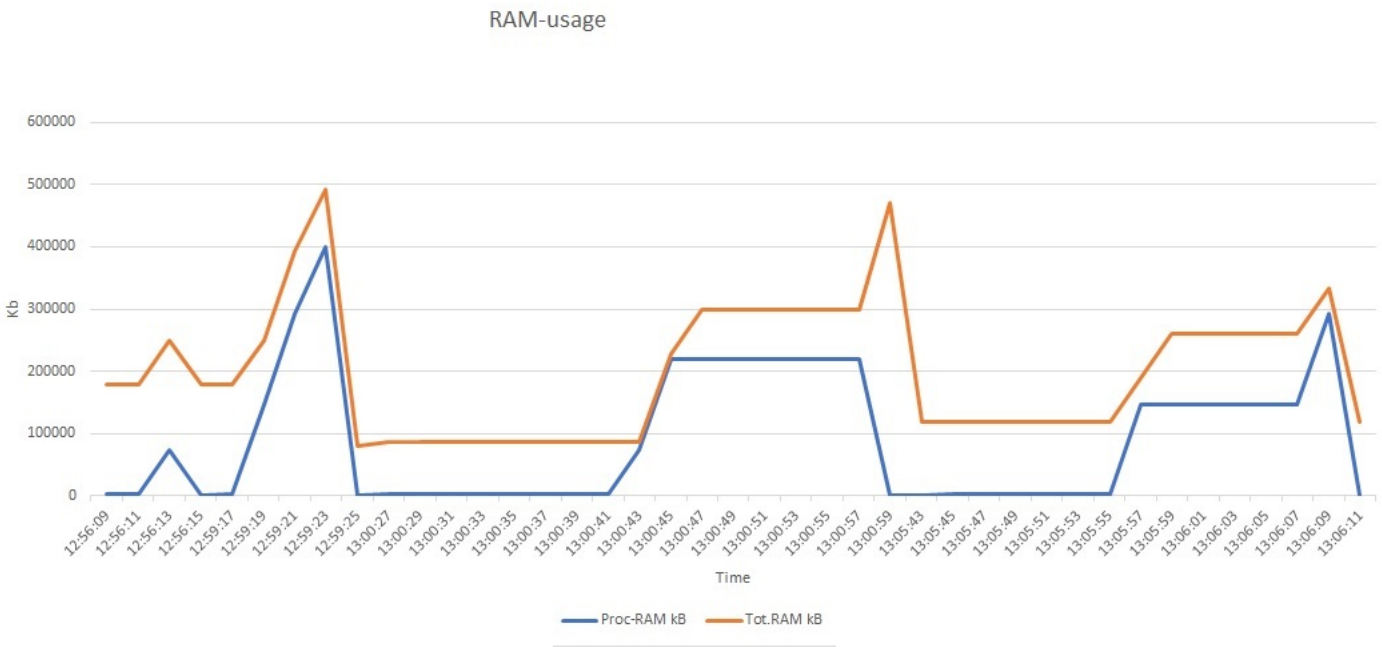
I Excel finns det flera olika diagram som kan användas för att presentera körningarna. Linjediagram har valts för dessa parametrar eftersom de beskriver hela körningen utefter tidsintervallet. Det möjliggör även jämförelse mellan processens värden och de totala värdena för hela systemet. I figur 6.2 visas körningsvärden för parametern CPU-användning för en process i ett linjediagram med procent på vertikala axeln samt tid på den horisontella axeln. Tidslinjen för processens CPU-användning syns i blått och för den totala användningen i orange. Processen har körts fyra gånger vid olika tidpunkter.




**Figur 6.2:** CPU-användning

Under första körningen gjordes endast en mindre minnesallokering vilket ej belastade processorn markant, vilket syns i diagrammet. Ur diagrammet kan utläsas att processens CPU-användning går upp till 20% vid 12:59:17 för den andra körningen. Detta beror på att flera minnesallokering genomförs i testprogrammet. Vid 13:00:27 genomför testprogrammet ett CPU-test där flera beräkningar sätts igång. Här ökar processens CPU-användning till cirka 38% och den totala CPU-användningen ligger runt 90%. Testet pågår till 13:00:39 och sedan sjunker CPU-användningen för både processen och totalt. Ytterligare tre CPU-tester görs under körningarnas gång. Vid 13:00:41, 13:05:43 samt 13:05:57. Ur diagrammet kan ses att dessa fyra tester visar snarlika resultat med ett fåtal avvikelser. Vid 13:00:57 görs flera minnesallokeringar vilket gör att processens CPU-användning stiger till cirka 12% samt 60% för totala CPU-användningen. Eftersom det sista testet avslutas med att hela körningen avslutar ger detta ett något annorlunda resultat.

I figur 6.3 beskrivs parametern RAM-användning i ett linjediagram med kB på vertikala axeln samt tid på den horisontella axeln. Tidslinjen för processens RAM-användning syns i blått och den totala RAM-användningen i orange. Detta är samma fyra körningar som beskrivits för CPU-användningen ovan. I början av processen förbrukar processen inget medan den totala användningen ligger på strax under 200000 kB som används till annat. Vid 12:56:11 görs ett test där 80000 kB allokeras för att sedan frigöras. Vid 12:59:17 görs fem allokeringar i rad vilket gör att processens användning stiger till 400000 kB. Här kan noteras att den totala användningen stiger i takt med processen men att processen även tar minne från totala användningen som Linux reserverat för snabb åtkomst. Därefter frigörs processens RAM. Ytterligare allokeringar görs, tre gånger vid 13:00:41 vilket gör att processens RAM-användning stiger till cirka 200000 kB, två allokeringar vid 13:05:55 samt två allokeringar vid 13:06:07 vilket gör att användningen stiger till strax under 300000 kB.




**Figur 6.3:** RAM-användning

# 7
# Slutsats

Detta kapitel behandlar diskussion och slutsatser utefter resultaten. Möjliga förbättringar som skulle kunna implementeras i fortsatta arbeten diskuteras och presenteras.

## 7.1 Diskussion

Efter att ha genomfört detta projekt kan det konstateras att det är möjligt att konstruera ett enkelt diagnostiseringsprogram för embedded Linux. Under resultatdelen kan utläsas att båda diagrammen visar att Dialux läser och presenterar korrekta värden då dessa följer testprogrammets direktiv. Det är dock svårt att skapa ett helt omfattande testprogram och därmed kan det finnas undantag där programmet inte presterar enligt uppsatta mål. Figur 6.1 visar att Dialux även sparar datan till en CSV-fil och kan på så sätt presenteras i ett Excel dokument vilket var ett av de uppsatta målen. Frågeställningarna har besvarats och motiverats utefter syftet med projektet. Dock skulle dessa kunna variera efter personlig preferens, men här har hänsyn främst tagits till vad som är genomförbart inom given tidsram och vad som anses passande enligt dialog med handledare på Diadrom.

Målet med arbetet samt programmets funktion och syfte har justerats ett flertal gånger under projektet gång. Detta eftersom nya kunskaper har erhållits och där både möjligheter och begränsningar hittats. Det gjorde att det tog en period innan det kunde fastställas vad som var genomförbart och hur det skulle kunna skapas. Till en början ansågs det lämpligt att projektet skulle innefatta fler parametrar att diagnostisera. Detta innefattade I/O-portar, antal trådar samt diagnostisera Ethernet porten. Under arbetets gång uteslöts dessa eftersom det ej ansågs tillföra något extra till syftet med projektet och att det var mer lämpligt att avgränsa det till två parametrar. Detta gjorde att mindre tid behövdes läggas på parametrar och istället fokusera på de centrala delarna som nämns i frågeställningarna.

Eftersom det finns olika lösningar och svar på frågeställningarna så har det krävts flera olika iterationer där olika alternativ testats. Programarkitekturen har också gjorts om ett antal gånger då det inte gett önskat resultat enligt syftet. Detta har gjort att mycket tid lagts på programmering och felsökning. Dock har detta gett mer underlag till motiveringen av val i frågeställningarna eftersom tydliga brister med vissa alternativ visat sig. Efter att sista iterationen skapats lades tid på att testa den och försöka finna felaktigheter.

Eftersom det finns många olika sätt att presentera data på så blev denna process utdragen. Först valdes XML där det tog lång tid att sätta sig in i hur det fungerar. Detta alternativ gav inte heller önskat resultat då värdena blev svårlästa och svårhanterade. Samma problem uppstod med JSON och där det även blev problem med parsingen. Detta gjorde att mycket tid lades på detta som istället kunde lagts på mer centrala uppgifter för projektet. Implementationen av en CSV-fil gick betydligt smidigare och i Excel blev det mer lätthanterligt där grafer kunde skapas. Därmed blev en viktig lärdom att inte lägga för mycket extraarbete på innehåll som inte anses centralt för projektet. Detta gav dock kunskaper inom XML och JSON som kan vara användbara samtidigt som det gav upphov till en motivering om varför CSV och Excel valdes.

Projektets innebörd ur ett perspektiv med fokus på hållbar utveckling är främst ekonomiskt. Att tillämpa Dialux för att diagnostisera processer i embedded Linux skulle kunna bidra till mer effektiva system. Detta då utvecklare kan använda programmet för att optimera processer som kör på embedded Linux, vilket skulle ge företag möjlighet till långsiktig ekonomisk lönsamhet. Mer effektiva system skulle även kunna bidra till en längre livscykel för enheterna. Vilket skulle kunna minska resursanvändningen. Detta ger även projektet en viss innebörd för miljöaspekten inom hållbar utveckling.

Projektet har innefattat många olika moment, det har gett en utökad förståelse för vad embedded Linux används till och hur det är uppbyggt. Detta innefattar bland annat linuxkärnan, de olika virtuella filsystemen samt kommandon. Att arbeta i samarbete med Diadrom har även gett en inblick i hur diagnostik utförs och vad som behövs tänkas på i processen när ett diagnostiseringsprogram framtas. Utöver detta har även tidigare kunskaper inom C-programmering utvecklats och utvidgas till att kunna integrera detta med Linux. I projektet har även C++ använts vilket har gett nya kunskaper.

## 7.2 Förslag till fortsatt arbete

Diagnostiseringsprogrammet som skapats är relativt enkelt och avgränsat. Många idéer och förslag har uppkommit under projektets gång som skulle kunnat implementerats om mer tid fanns. Detta skulle istället kunna utföras i ett fortsatt arbete. Dels skulle fler parametrar kunna inkluderas. Det finns flertalet parametrar utöver de som presenterats i denna rapport som skulle vara intressanta att diagnostisera. Värden för många parametrar skulle kunna hämtas via proc likt det som gjorts i detta projekt.

Detta projekt är avgränsat till att enbart spara och presentera värden för processen i en CSV-fil. Här skulle även en realtids funktion kunna implementeras där en varning uppkommer om någon parameter överskrider eller avviker ett visst intervall. Detta skulle ge en ytterligare dimension till diagnostiseringsprogrammet och ge möjlighet för användaren att notera en avvikelse under körning istället för vid analysen efter körningen.

För att utöka programmet ytterligare skulle ett gränssnitt kunna skapas. Detta skulle exempelvis kunna vara i form av en .exe fil. Därmed skulle användaren vid uppstart av Dialux kunna få välja olika alternativ för hur diagnostiseringen ska utföras. Här skulle användaren kunna få skriva in vilket program som ska diagnostiseras och sedan fylla i vilka parametrar som ska användas och vid vilka avvikelser som en varning ska uppkomma. En stor fördel med detta vore att användaren då inte behöver ändra i Dialux källkod och på så vis även mer lättanvänt.

En ytterligare möjlighet skulle vara att lagra den insamlade datan i en molnstjänst. En tjänst som nämnts under projektet är OpenCensus [24]. Detta skulle göra programmet mer flexibelt där åtkomst till datan kan ske från olika enheter. Detta ger också en utökad säkerhet då datan inte är beroende av en enhet utan alltid finns lagrad i molnet. Med OpenCensus finns även många möjligheter att analysera samt presentera datan. Dessa funktioner skulle göra Dialux till ett mer omfattande och användbart program.

# Referenser


[1] J. Fiddler, "Is embedded linux the right answer?" *EEtimes*, 2012. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1206531# [Hämtad2019/04/21]

[2] Diadrom, "Om oss," Available at https://diadrom.se/om-oss [Hämtad 2019/04/21].

[3] Okänt, "Beaglebone black, hämtad enligt creative commons attribution-share alike 3.0 license," Available at https://beagleboard.org/black [Hämtad 2019/04/22].

[4] ——, "Om debian," Available at https://www.debian.org/intro/about [Hämtad 2019/06/11].

[5] G. B.-Y. P. G. Karim Yaghmour, Jon Masters, *Building embedded Linux systems*, 2nd ed. 1005 Gravenstein Highway North, Sebastopol,CA: O'Reilly Media, 2008. [Online]. Available: https://www.dawsonera.com/readonline/9780596154882/startPage/55/1

[6] Okänd, "Calculating cpu utilization," 2002. [Online]. Available: http://ibmsystemsmag.com/ibmi/administrator/performance/calculating-cpu-utilization/

[7] C. Simmonds, *Mastering Embedded Linux Programming*, 2nd ed. Livery Place 35 Livery Street, Birmingham, UK.: Packt Publishing, 2017. [Online]. Available: https://app.knovel.com/web/toc.v/cid:kpMELPE006/viewerType:toc//root_slug:mastering-embedded-linux/url_slug:managing-memory?=undefined&issue_id=kpMELPE006&hierarchy=[Hämtad2019/04/24].

[8] A. C.-M. N. A. B. Daniel Quinlan, Michael Kerrisk, "Linux programmer's manual," Available at http://man7.org/linux/man-pages/man5/proc.5.html [Hämtad 2019/05/14].

[9] R. Chopra, *C Programming: A Self-Teaching Introduction*, 1st ed. 22841 Quicksilver Drive Dulles, VA, 20166 UK.: David Pallai, 2018. [Online]. Available: https://library-books24x7-com.proxy.lib.chalmers.se/assetviewer.aspx?bookid=128095&chunkid=372912526&rowid=16[Hämtad2019/04/24].

[10] F. Y. Li Zheng, Yuan Dong, *C++ Programming*, 1st ed. Tsinghua University, Beijing, China.: De Gruyter, 2019. [Online]. Available: https://www.degruyter.com/view/books/9783110471977/9783110471977-001/9783110471977-001.xml

[11] B. Lantz, *Machine Learning with R*, 2nd ed. Birmingham, UK: Packt Publishing, 2016. [Online]. Available: https://app.knovel.com/web/toc.v/cid:kpMLREDHB3/viewerType:toc//root_slug:machine-learning-with/url_slug:importing-saving-data?=undefined&issue_id=kt0113PGQB&hierarchy=[Hämtad2019/05/05].

[12] G. Coley, “System reference manual,” Available at https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual [Hämtad 2019/06/11].

[13] J. Skansholm, *C från början*, 1st ed. Åkergränden 1, Lund, Sverige: Studentlitteratur AB, 2016 ISBN: 9789144114583.

[14] A. M. Mark Kraeling, *Software Engineering for Embedded Systems*, 1st ed. 225 Wyman Street, Waltham, MA 02451, USA: Elsevier Inc, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124159174000256

[15] R. E. McShea, *Test and Evaluation of Aircraft Avionics and Weapon Systems*, 1st ed. 911 PaverstoneDrive,Suite B, Raleigh,NC: SciTechPublishing, 2010. [Online]. Available: https://app.knovel.com/web/toc.v/cid:kpTEAAWS04/viewerType:toc//root_slug:test-evaluation-aircraft/url_slug:electronic-warfare-systems?=undefined&issue_id=kt008RZAV5&hierarchy=[Hämtad2019/05/07]

[16] C. B. Wim Torfs, “Tdma on commercial of-the-shelf hardware: Fact and fiction revealed,” Amsterdam, Nederländerna, Tech. Rep., 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1434841115000254?via%3Dihub[Hämtad2019/05/14]

[17] J. L. B. Marta Beltran, Antonio Guzman, “A new cpu availability prediction model for time-shared systems,” Piscataway, New Jersey, USA, Tech. Rep., 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4445661

[18] J. B. Karla Saur, “Locating Œ86 paging structures in memory images,” Amsterdam, Nederländerna, Tech. Rep., 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S174228761000054X

[19] Okänt, “Cortex-a8 technical reference manual,” Available at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.a8/index.html [Hämtad 2019/05/18].

[20] D. A. Joe Fawcett, Liam R.E. Quin, *Beginning XML*, 5th ed. Birmingham, UK: Wrox Press, 2012. [Online]. Available: https://library-books24x7-com.proxy.lib.chalmers.se/toc.aspx?bkid=46607

[21] J. Friesen, *Java XML and JSON*, 1st ed. New York, New York, USA: Apress Media, 2016. [Online]. Available: https://library-books24x7-com.proxy.lib.chalmers.se/assetviewer.aspx?bookid=115814&chunkid=458156494&rowid=245

[22] M. K. Thomas Koenig, “Linux programmer’s manual - system(3),” Available at http://man7.org/linux/man-pages/man3/system.3.html [Hämtad 2019/05/20].

[23] M. Kerrisk, “top - display linux processes,” Available at http://man7.org/linux/man-pages/man1/top.1.html [Hämtad 2019/06/11].

[24] Okänd, “What is opencensus?” Available at https://opencensus.io/ [Hämtad 2019/06/11].