Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

OSEK-kompatibilitet hos ENEA OSE_{ck}

av

Jenny Palmberg
Lili Ren

LIU-IDA-EX—08/049--SE

2008-10-29

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Examensarbete

OSEK-kompatibilitet hos ENEA OSE$_{ck}$

av

Jenny Palmberg
Lili Ren

LITH-IDA-EX--2008/049--SE

2008-10-29

Examensarbete

OSEK-kompatibilitet hos ENEA OSE$_{ck}$

av

Jenny Palmberg
Lili Ren

LITH-IDA-EX--2008/049--SE

Handledare : **Andreas Edlund**

vid Enea Services Linköping AB

Examinator : **Professor Petru Eles**
Institutionen för datavetenskap
vid Linköpings universitet

Sammanfattning

Målet med examensarbetet var att undersöka om det var möjligt att genom ett kompatibilitetsbibliotek se till att Eneas realtidsoperativsystem OSE_{ck} kan uppfylla kraven i operativsystemsstandarden OSEK.

OSE_{ck} visade sig tillhandahålla all efterfrågad funktionalitet och ett kompatibilitetsbibliotek som innehöll OSEK's API kunde därmed implementeras. Ett verktyg togs fram för att utifrån en fil, innehållandes objekt beskrivna i OSEK's konfigurationsspråk OIL, plocka ut den information som behövdes för att konfigurera både OSE_{ck} och OSEK.

Slutsatsen av examensarbetet blev att det gick att göra OSE_{ck} OSEK-kompatibelt genom ett yttre lager och att inga ändringar i OSE_{ck}'s kärna var nödvändiga. Givetvis påverkar lagret operativsystemets prestanda negativt men det får ändå anses att dess prestanda fortfarande är så pass bra att en integration i OSE_{ck}'s kärna ej behövs.

För att ett operativsystem ska kunna göras OSEK-kompatibelt måste det ha prioritetsbaserad schemaläggning samt att task som blir avbrutna hamnar först i sin prioritetskö. Dessutom måste det vara möjligt att exekvera kod precis innan ett task börjar köra för första gången eftersom det ska finnas stöd för en PreTaskHook.

Nyckelord : Realtidsoperativsystem, OSEK

Förord

Rapporten handlar om anpassningen av ett realtidsoperativsystem till en befintlig standard. Den är det sista momentet i det examensarbete som har utförts på utbildningen Teknisk Fysik & Elektroteknik (Y) på Linköpings Tekniska Högskola. Examensarbetet har genomförts på Enea Linköping Services AB med stöd från institutionen IDA på Linköpings Tekniska Högskola.

Ett särskillt tack riktas till de som har gett stöd och handledning under arbetets gång.

Innehåll

1 Inledning **1**
1.1 Bakgrund 1
1.2 Mål 2
1.3 Rapportens disposition 2
1.4 Avgränsningar 3
1.5 Utvecklingsmiljöer 3
1.6 Förtydligande 4

2 OSEK/VDX **5**
2.1 Bakgrund och Målsättning 5
2.2 Områden 6
2.3 Objekt 6

3 OSEK Implementation Language (OIL) **8**
3.1 Bakgrund 8
3.2 Beskrivning 8
3.3 Implementations- och Applikationsdefinition 9
3.4 Exempel på en OIL-applikation 10

4 OSEK OS **12**
4.1 OSEK OS Arkitektur 12
4.2 Taskhantering 13
4.2.1 Basic Task och Extended Task 13
4.2.2 Schemaläggningsprinciper 15

4.2.3 Schemaläggare 15
4.2.4 Exempel 16
4.3 ISR 18
4.4 Resurshantering 19
4.4.1 Prioritetsinversion 19
4.4.2 Priority Ceiling Protocol (PCP) 20
4.4.3 PCP som lösning på prioritetsinversion 22
4.4.4 Interna resurser 23
4.4.5 Exempel 24
4.5 Event 25
4.5.1 Exempel 26
4.6 Alarm och Counter 28
4.6.1 Exempel 30
4.7 Felhantering 32
4.7.1 Standard / Utökad 32
4.8 Hook-rutiner 33
4.9 Application mode 33
4.10 Conformance Classes 34

5 OSE RTOS **36**

6 OSE$_{ck}$ **38**
6.1 Konfiguration av OSE$_{ck}$ 38
6.2 Initiering 39
6.3 Processer 39
6.3.1 Prioriteter 40
6.3.2 Tillstånd 40
6.4 Schemaläggare 41

7 OIL-tool **42**
7.1 Parsning 44

8 Implementation av OSEK lager **46**
8.1 Task-anpassning 46
8.2 Resurs-anpassning 47
8.2.1 Interna resurser 48

8.2.2 Schemaläggare som resurs 48
8.3 Alarm- och Counter-anpassning 48
8.4 Event-anpassning 49
8.5 ISR-anpassning 49
8.6 Hook rutiner 50
8.6.1 StartupHook 50
8.6.2 PreTaskHook och PostTaskHook 50
8.6.3 ErrorHook 51
8.7 Schemaläggare 51
8.8 Conformance Classes 52

9 Prestanda 53
9.1 Förutsättningar 53
9.2 Jämförelse av storlek 53
9.2.1 Storlek på OSEK-lager 54
9.2.2 Storlek på objektfiler 56
9.2.3 Description och Control Block listor 57
9.3 Tidmätningar 59
9.3.1 Initiering 59
9.3.2 OSEK's API-anrop 60

10 Diskussion 67
10.1 Krav på underliggande operativsystem 67
10.2 Ändringar i kärnan 68
10.3 Utvidgning och förbättringsförslag 68

11 Andra OSEK kompatibla OS 71
Referenser 73

Litteraturförteckning 73

Kapitel 1

Inledning

1.1 Bakgrund

Innan datoriseringen byggdes fordonens olika delar ofta helt isolerat från varandra, till exempel utvecklingen av bromssystemet var helt skild från den av bränsleinsprutningen. När mekanik och analog logik sedan började ersättas av mjukvara inom bilindustrin, levde de gamla mekaniska traditionerna kvar och ny funktionalitet utvecklades fortfarande i isolation. Detta ledde till att det blev en uppsjö av olika processorer och operativsystem i fordonen och i en bransch där man jagar ören är detta givetvis väldigt kostnadsineffektivt.

Med dagens ökade krav på komplexa lösningar, där bland annat nätverksbaserade komponenter krävs, så erfodras en samordnad utvecklingsprocess. AUTOSAR är ett samarbetsprojekt mellan flera automotivföretag vars mål är att skapa en standard inom branschen för hur en mjukvaruplattform ska byggas. Detta ska öka möjligheten att plocka överliggande mjukvarukomponenter från olika tillverkare och få dem att fungera tillsammans. Många av lågnivåkomponenterna i AUTOSAR, som realtidsoperativsystemet och kommunikationsmiljön, kommer från OSEK/VDX som är en standard för operativsystem avsedda för distribuerade realtidssystem i motorfordon.

Ett svenskt initiativ till att ta fram en AUTOSAR-kompatibel plattform har fått namnet SWAP, och det finns nu planer på att använda Eneas operativsystem OSE$_{ck}$ som en av komponenterna i SWAP. Då AUTOSAR-standarden innehåller kravet att det underliggande operativsystemet ska vara OSEK-kompatibelt krävs en anpassning av OSE$_{ck}$ till OSEK-standarden.

1.2 Mål

Målsättningen med examensarbetet är att undersöka om det är möjligt att skapa ett bibliotek ovanpå OSE$_{ck}$, så att det uppfyller kraven i OSEK-standarden. Visar sig detta vara möjligt ska ett sådant bibliotek implementeras.

En utredning över vad som krävs av ett operativsystem för att göra det OSEK-kompatiblet ska göras. Om OSE$_{ck}$ saknar stöd för någon av funktionaliteterna som krävs för att implementera OSEK, så ska eventuellt ändringar i dess kärna göras.

Prestandamätningar ska göras för att undersöka hur kompatibilitetsbiblioteket och eventuella ändringar i kärnan påverkar operativsystemets prestanda.

1.3 Rapportens disposition

Kapitel 2 till och med kapitel 4 beskriver de, för examensarbetet, väsentliga delarna av operativsystemstandarden OSEK.

I kapitel 5 till och med kapitel 6 finns en kort genomgång av Eneas realtidsoperativsystem. Först ges en ytlig beskrivning av de tre olika varianterna av operativsystemet OSE. Därefter ges en något mer grundlig genomgång av OSE$_{ck}$ som är den variant av OSE som har använts i examensarbetet.

Kapitel 7 och 8 redogör för hur anpassningen av OSE$_{ck}$ till OSEK gick till.

Först beskrivs det hur ett verktyg togs fram för att från OIL-filer plocka ut den information som behövs för att konfigurera OSE_{ck} och OSEK. Sedan görs en jämförelse mellan OSE_{ck} och OSEK och det redogörs, utifrån deras likheter och skillnader, för hur OSEK-lagret har implementerats.

Kapitel 9 innehåller tabeller med resultatet från de prestandamätningar som har utförts. Till tabellerna finns även förklaringar av hur mätningarna har gått till, varför de har gjorts och varför de ser ut som de gör.

I kapitel 11 finns en kort presentation av några OSEK-kompatibla operativsystem.

1.4 Avgränsningar

I examensarbetet har endast en anpassning till OSEK OS gjorts. Kapitlet i OSEK-specifikationen som rör meddelanden tillhör kommunikationsdelen av OSEK och har därför utelämnats. OIL-tool är dock implementerat på ett sådant sätt att det ska vara enkelt att lägga till ny funktionalitet.

1.5 Utvecklingsmiljöer

Utvecklingen har skett i en Windows-miljö där gcc har använts som kompilator under cygwin. Både OIL-tool och alla OSEK's systemfunktioner är implementerade i C.

ANTLR är en parsergenerator som har använts för att skriva en parser för konfigurationsspråket OIL.

Systemet har testats både på en SFK (Soft Kernel) och på två olika utvecklingskort. Att det var möjligt att testa systemet på en SFK underlättade utveckling och felsökningen betydligt.

I examensarbetet ingick det dock även att systemet skulle köras felfritt på tilltänkt hårdvara och det har därför testats på två utvecklingskort med

olika processorer. Till att börja med kördes systemet på en processor med mycket lite minne och det kunde därför inte testas fullt ut. Därför beställdes ny hårdvara med mer minne där all funktionalitet fick plats och kunde testas. Prestandatesterna genomfördes på den senare processorn som var av typen MPC5554.

1.6 Förtydligande

Som ett stöd, till läsaren av den här rapporten, ska ett par begrepp redas ut lite tydligare. OSE är ett realtidsoperativsystem som Enea har utvecklat och OSE$_{ck}$ är en variant av OSE som ska anpassas till en standard för realtidsoperativsystem kallad OSEK. Observera att OSEK alltså inte är en Enea produkt, även om namngivningen är mycket lika, utan är den standard för realtidsoperativsystem som samarbetsprojektet AUTOSAR använder.

Kapitel 2

OSEK/VDX

2.1 Bakgrund och Målsättning

OSEK grundades 1993 i ett samarbetsprojekt mellan flera tyska automotivföretag. Syftet var att försöka skapa en industristandard för öppen arkitektur för distribuerade kontrollenheter i fordon. 1994 slogs det tyska projektet ihop med ett liknande franskt projekt inom automotivbranschen, VDX. För enkelhetens skull kommer OSEK/VDX förkortas till OSEK i rapporten.

Målsättningen med OSEK är att skapa portabilitet och återanvändbarhet av applikationsmjukvara genom att ha ett applikationsoberoende gränssnitt. Gränssnittet ska även vara hårdvaru- och nätverksoberoende. Dessutom ska OSEK arkitekturen vara skalbar för att kunna anpassas efter system med olika krav på till exempel minnesanvändning. [9]

2.2 Områden

OSEK är uppdelat i huvudområdena OSEK Operating System (OS), OSEK Communication (COM) och OSEK Network Management (NM). Examensarbetet berör bara OSEK OS och konfigurationsspråket, OIL, som OSEK använder för konfiguration av bland annat operativsystemet. [9]

2.3 Objekt

OSEK definierar ett antal systemobjekt, med tillhörande attribut och objektreferenser, och ett API för att använda dessa. Exempel på systemobjekt är TASK, RESOURCE och ALARM och exempel på systemanrop som påverkar dessa är ActivateTask, GetResource och CancelAlarm. De objekt som tillhör OSEK OS beskrivs utförligt senare i rapporten. Dessutom beskrivs det hur en del av funktionerna som ingår i OSEK's API har implementerats. Nedan finns en tabell med OS-objekt och API. [14]

Objekt	**API**
TASK	ActivateTask TerminateTask ChainTask GetTaskID GetTaskState Schedule
EVENT	SetEvent ClearEvent GetEvent WaitEvent

Objekt	**API**
ISR	EnableAllInterrupts DisableAllInterrupts ResumeAllInterrupts SuspendAllInterrupts ResumeOSInterrupts SuspendOSInterrupts
RESOURCE	GetResource ReleaseResource

Objekt	Funktioner
ALARM	GetAlarmBase GetAlarm SetRelAlarm SetAbsAlarm CancelAlarm
OS	GetApplicationMode StartOS ShutdownOS ErrorHook PreTaskHook PostTaskHook StartupHook ShutdownHook

Tabell 2.2: Tabellen innehåller OS-objekt med tillhörande funktioner som ingår i OSEK's API.

Kapitel 3

OSEK Implementation Language (OIL)

3.1 Bakgrund

Målet med OSEK är att tillhandahålla en standard som ska öka kompatibilitet och portabilitet av mjukvaruapplikationer. Ett hjälpmedel för att åstadkomma detta är konfigurationsspråket OIL som ingår i OSEK-standarden [3].

3.2 Beskrivning

Vilka OSEK-objekt som ska finnas med i en applikation definieras statiskt genom att alla objekt och deras statiska attribut och referenser beskrivs med hjälp av objekt i OIL. Alla objekt finns definierade med sina standardattribut i OIL-specifikationen [3]. Det är inte tillåtet att skapa nya typer av objekt men implementationsspecifika attribut till de tillåtna objekten kan däremot läggas till.

Att objekten och många av deras attribut definieras statiskt medför att

schemaläggning av systemet kan göras redan i utvecklingsfasen. Till exempel priority ceiling protokollet kan användas tack vare att alla prioriteter på task är definierade redan vid uppstart och protokollet är en mycket viktig del i hur OSEK hanterar taskschemaläggning. Detta begrepp återkommer och förklaras senare i rapporten, se avsnitt 4.4. [3]

3.3 Implementations- och Applikationsdefinition

En OIL-konfigurationsfil är uppdelad i två delar, en implementationsdefinition och en applikationsdefinition. Implementationsdefinitionen specificerar hur OIL-objekten är uppbyggda, det vill säga vilka attribut de kan ha och av vilken typ attributen är. OIL-standarden definierar vilka attribut som måste finnas med och vilka som är valfria. Implementationsdefinitionen innehåller även gränsvärden eller standardvärden för vissa av attributen. Det kan bara finnas en definition för varje objekttyp och attribut i implementationsdefinitionen.

I applikationsdelen av OIL-konfigurationsfilen definieras hur många objekt det ska finnas av en viss objekttyp. Objektattribut ges ett värde och eventuella referenser tilldelas namnet på objektet som refereras. Det kan finnas flera objekt av samma typ och ibland även flera attribut av samma typ i applikationsdefinitionen. Enda undantaget är CPU-objektet som innehåller alla andra objekt och som det endast kan finnas ett av. Det finns en Implementations- och Applikationsdefinition för varje CPU is systemet. [3]

3.4 Exempel på en OIL-applikation

OIL har en lite C-lik syntax med klammerparenteser och semikolon som avdelare. Nedan visas först exempel på hur implementationsdefinitionen för ett os- och task-objekt kan se ut och sedan visas exempel på applikationsdefinitionen för samma objekt.

```
IMPLEMENTATION OSEK
{
  OS
   {
    BOOLEAN STARTUPHOOK = FALSE;
    BOOLEAN ERRORHOOK = FALSE;
    BOOLEAN SHUTDOWNHOOK = FALSE;
    BOOLEAN PRETASKHOOK = FALSE;
    BOOLEAN POSTTASKHOOK = FALSE;
   };

  TASK
   {
    UINT32 [0 .. 31] PRIORITY;
    RESOURCE_TYPE RESOURCE[];
    UINT32 ACTIVATION = 1;
    ENUM [
          NON,
          FULL
        ] SCHEDULE;
   };
};
```

Figur 3.1: Implementationsdefinition

I OS-objektet specificeras att objektet endast får innehålla fem attribut, vilka dessa är och vilka standardvärden de har. Attributen i det här exemplet är sanningsvärden för vilka av OSEK's hookrutiner som ska användas. I taskobjektet specificeras bland annat att prioriteten för ett task måste vara mellan 0 och 31. Taskobjektet innehåller även en enum som talar om att det task som objektet representerar, kan vara antingen preemtive

[FULL] eller non preemtive [NON].

```
CPU OSEK
{
  OS OSEck
   {
    STARTUPHOOK = TRUE;
    ERRORHOOK = TRUE;
    SHUTDOWNHOOK = FALSE;
   };

 TASK extended_preemt_task
  {
    PRIORITY = 15;
    ACTIVATION = 3;
    SCHEDULE = FULL;
    RESOURCE = resource_1;
  };
};
```

Figur 3.2: Applikationsdefinition

Objektet CPU innehåller alla andra objekt och det måste alltid finnas just ett CPU-objekt i en OSEK-applikation. I OS-objektet sätts några av attributen medan de andra får de standardvärden de blev tilldelade i implementationsdefinitionen. Det definieras att det task som objektet beskriver, får aktiveras tre gånger och har prioritet 15 samt att det är preemtive eftersom SCHEDULE = FULL. Referensen till en resurs är intressant att notera och varför den finns där kommer att beskrivas i stycke 4.4.

Kapitel 4

OSEK OS

4.1 OSEK OS Arkitektur

OSEK OS tillhandahåller ett API vars funktioner listas i tabell 2.2. Endast task, som schemaläggs av schemaläggaren, och interrupt, som bestäms av hårdvaran, kan använda sig av dessa funktioner. Det är endast task och interrupt som får processortid.

OSEK har tre prioritetsnivåer för interrupt, schemaläggare och task. Längst ner på skalan finns tasknivån som i sin tur kan bli indelad i en diskret skala av taskprioriteter där högst nummer är lika med högst prioritet. Det kan finnas flera task per prioritet.

Prioritetsnivån för interrupt är högre än den för task. Det kan finnas flera prioritetsnivåer för interrupt och även flera interrupt per prioritetsnivå. Hur prioritetsindelning för interrupt sker är hårdvaru- och implementationsspecifikt.

Det finns också en logisk nivå för schemaläggaren som kan sägas ligga mellan task och interrupt nivåerna, men den har ingen egentlig prioritet utan det är bara ett logiskt koncept. [14]

4.2 Taskhantering

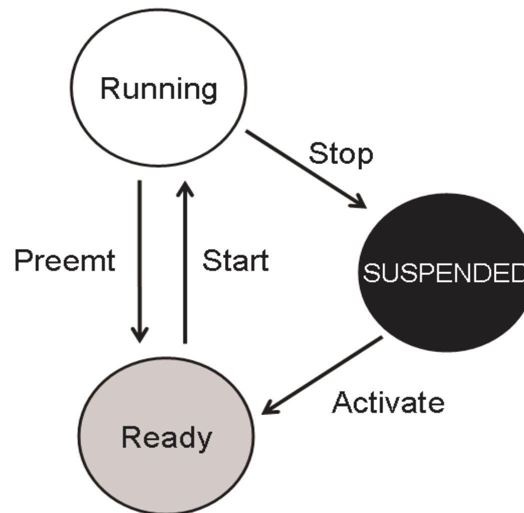
I OSEK används begreppet task för att beskriva hur ett program delas upp i delar med olika uppgifter. Olika programdelar, det vill säga task, har olika prioritet och det är schemaläggaren som bestämmer i viken ordning programdelarna ska exekveras.

Endast ett task per processor kan exekveras åt gången och detta task sägs vara i tillståndet running. Alla task som är redo att köra men ändå inte får tillgång till processorn är i tillståndet ready. Ibland måste ordningen, som koden exekveras i, synkroniseras och detta görs genom att task kan vänta på event och därmed hamna i tillståndet waiting. Event beskrivs i stycke 4.5. Ett task kan även vara suspended, vilket betyder att det är passivt medan det väntar på att bli ready.

4.2.1 Basic Task och Extended Task

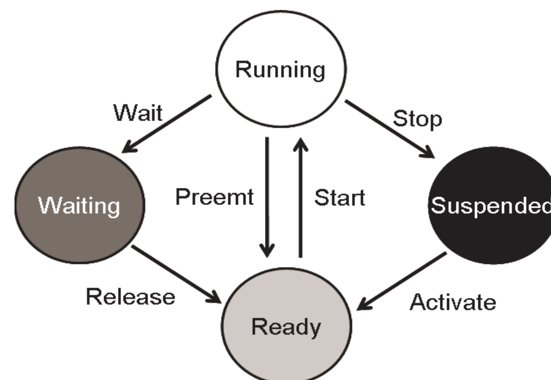
Det finns två typer av task i OSEK, basic task (BT) och extended task (ET). Skillnaden mellan dessa är att endast extended task får använda sig av OSEK-anropet WaitEvent och detta leder till en viss skillnad i tillståndsgraferna för tasken.

Basic task kan hamna i tillstånden running, ready eller suspended. De släpper bara processorn om de terminerar eller blir avbrutna av ett högre prioriterat task eller interrupt.



Figur 4.1: Tillståndsgraf för Basic Task

Tillståndsgrafen för extended task innehåller samma tillstånd som grafen för basic task. Extended task kan dock hamna i ytterligare ett tillstånd genom att anropa WaitEvent som orsakar att tasket går över till att vara waiting. Detta betyder att ett extended task självmant kan välja att släppa processorn och låter på så vis lägre prioriterade task köra medan det väntar på ett event.



Figur 4.2: Tillståndsgraf för Extended Task

4.2.2 Schemaläggningsprinciper

I OIL-filen definieras det om ett task får avbrytas under sin exekvering eller inte. Task är antingen helt preemtive, det vill säga de får avbrytas när som helst under sin exekvering av högre prioriterade task, eller helt non preemtive och får då inte avbrytas när de väl har påbörjat sin exekvering. Både preemtive och non preemtive task kan dock när som helst bli avbrutna av interrupts.

En applikation kan använda sig av antingen preemtive, non preemtive eller mixad schemaläggning. Innehåller en applikation enbart preemtive task är schemaläggningsprincipen preemtive och om det endast finns non preemtive task är den non preemtive. Vid en blandning av de olika typerna av task tillämpas mixad schemaläggning.

4.2.3 Schemaläggare

Schemaläggaren har hand om när schemaläggning ska ske och vilket task som ska få processortid. Beroende på om det är full eller non preemtive schemaläggning kan olika schemaläggningspunkter identifieras.

Schemaläggningen vid full preemtive schemaläggning sker när ett task aktiveras eller när ett task termineras. Om ett task väljer att vänta på ett event eller om ett event sätts för ett task ska också schemaläggning göras. När ett task släpper en resurs kan dess prioritet ändras och även här ska schemaläggaren kontrollera vilket task som ska ha processortid.

Vid non preemtive schemaläggning är schemaläggningspunkterna något färre eftersom ett non preemtive task inte får avbrytas oavsett om högre prioriterade task blir ready eller inte under dess exekvering. Schemaläggning sker då när ett task avslutar sin exekvering eller om det börjar vänta på ett event. Ett non preemtive task kan även anropa OSEK-funktionen Schedule för att självmant släppa processorn och orsaka schemaläggning.

Schemaläggaren låter i första hand det task som har högst prioritet, det vill säga högst siffra, få processortid. Av task på samma prioritetsnivå, är

det det task som har varit ready längst som är först i kön för att få börja köra. Undantag från den regeln är om ett task blir avbrutet eller om det går över till ready från waiting. Då hamnar det först i sin prioritetskö. [14]

4.2.4 Exempel

```
TASK basic_preemt_h
{
   AUTOSTART = TRUE
   {
     APPMODE = OSDEFAULTAPPMODE;
   };

   PRIORITY = 30;
   SCHEDULE = FULL;
   ACTIVATION = 1;
};

TASK basic_non_preemt_m
{
   AUTOSTART = FALSE;
   PRIORITY = 15;
   SCHEDULE = NON;
   ACTIVATION = 1;
};
```

Figur 4.3: OIL-definition för två task

I figuren ovan ges ett exempel på hur task kan definieras som OIL-objekt. Task basic_preemt_h startas automatiskt när operativsystemet har startat och programmet kan börja köra om APPMODE (application mode) är lika med OSEDEFAULTAPPMODE. Vilket applikationstillstånd som operativsystemet befinner sig i definieras vid uppstart. Basic_preemt_h är ett preemtive task med prioritet 30.

Task basic_non_preemt_m är non preemtive eftersom SCHEDULE = NON

och har prioritet 15 vilket är lägre än prioriteten för basic_preemt_h.

Nedan visas ett exempel i pseuvdokod på hur några av OSEK's systemfunktioner används i de task som definierades av OIL-objekten ovan.

```
TASK(basic_preemt_h)
{
     StatusType status = 0;
     TaskStateType task_state;

     status = ActivateTask(basic_non_preemt_m);
     GetTaskState(basic_non_preemt_m, &task_state);

     if (status == E_OK && task_state == READY)
          TerminateTask();
     else for(;;);
}

TASK(basic_non_preemt_m)
{
     ActivateTask(basic_preemt_h);
     Schedule();
}
```

Figur 4.4: Exempelapplikation för task

I exemplet startas som sagt basic_preemt_h när programmet börjar köra. Det aktiverar basic_non_preemt_m och fortsätter sedan sin exekvering eftersom det har högst prioritet. För att försäkra att basic_non_preemt_m verkligen är redo att köra hämtas det i vilket state basic_non_preemt_m befinner sig genom ett anrop till GetTaskState. Därefter kontrolleras att task_state verkligen är READY och först när det är säkert att ett annat task är redo att köra går basic_preemt_h över i tillståndet suspended genom ett anrop till TerminateTask.

Basic_non_preemt_m får nu exekvera och aktiverar återigen basic_preemt_h med sin första instruktion ActivateTask. Eftersom basic_non_preemt_m är

non preemtive kan det inte avbrytas ens av högre prioriterade task och får därför fortsätta köra. Med anropet till Schedule begär basic_non_preemt_m själv att en schemaläggning ska ske och basic_preemt_h kan på nytt påbörja sin exekvering.

När instruktionen ActivateTask(basic_non_preemt_m) kör kommer en felkod returneras eftersom basic_preemt_m endast får aktiveras en gång i taget enligt OIL-objektet och det har ännu inte avslutat sin första aktivering. Det betyder att state inte är lika med E_OK och i just det här exemplet fastnar programmet i en oändlig loop.

Exempelkoden innehåller en del överflödig kod i basic_preemt_h mest för att demonstrera hur några av OSEK-funktionerna kan användas. Den kunde till exempel ha förenklats till följande:

```
TASK(basic_preemt_h)
{
	ChainTask(basic_non_preemt_m);
	for(;;);
}
```

Figur 4.5: Exempelapplikation för task

ChainTask avslutar först det task som anropar funktionen och aktiverar sedan ett nytt task. ChainTask returnerar endast om något fel uppstår vid termineringen av basic_preemt_h eller vid aktiveringen av basic_non_preemt_m. Om det är fallet fastnar programmet i exemplet återigen i en oändlig loop.

4.3 ISR

Som tidigare har nämnts definieras task- och interrupt-prioriteter statiskt i en OIL-fil. Interrupt har högre prioritet än task och kan avbryta både preemtable och non preemtable task när som helst under deras exekvering.

OSEK specificerar två typer av interrupt som kräver olika mycket i overhead. Interrupt av kategori 1 får inte använda OSEK's API och tar minst i overhead. Exekveringen, av ett task som har blivit avbrutet av ett kategori 1 interrupt, fortsätter i exakt samma punkt som den blev avbruten. Kategori 2 interrupt får däremot anropa vissa funktionerna som ingår i API. Schemaläggning sker inte i en ISR utan först efter att ett interrupt av kategori 2 har kört klart. [14]

4.4 Resurshantering

Delade resurser, såsom minnesareor och programsekvenser, kan orsaka svårförutsägbara fel om inte samtidig åtkomst av dessa undviks. OSEK använder sig av ett speciellt resursbegrepp för att lösa detta problem. De resurser som kommer att diskuteras nedan är ett logiskt begrepp som beskrivs med OSEK-objekt och är alltså inte sådana fysiska resurser som har nämnts ovan. I OSEK's API ingår det två funktionsanrop för att ta och släppa en resurs.

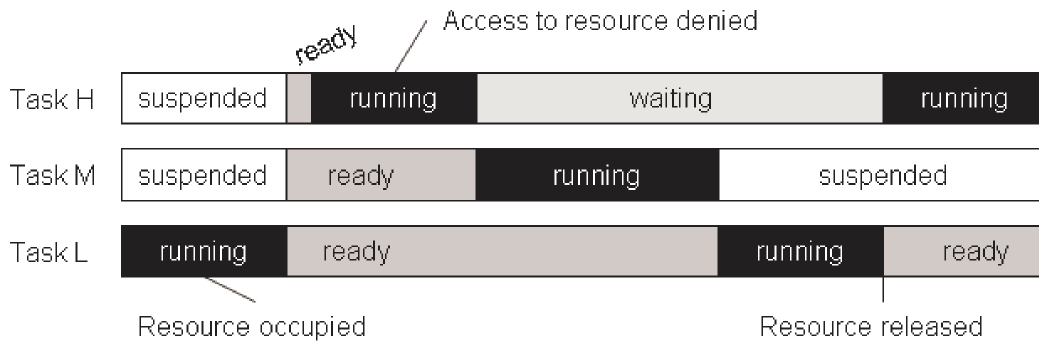
Om det inte finns några restriktioner för hur resurserna används, förutom att endast ett task i taget har åtkomst till dem, finns det risk för att till exempel prioritetsinversion eller deadlock uppstår. OSEK använder sig av Priority Ceiling Protocol (PCP) för att förhindra detta. Dessutom ska det kunna garanteras att två task aldrig håller samma resurs samtidigt och att task aldrig ska kunna hamna i ett tillstånd där de väntar på att få ta en resurs.

4.4.1 Prioritetsinversion

Ett problem som kan uppstå vid hantering av lås är prioritetsinversion. Prioritetsinversion uppstår när ett högre prioriterat task tvingas vänta på grund av att det vill ta en resurs som ett lägre prioriterat task håller. Enklast visas detta med ett exempel.

Exempel: Det finns tre preemtable task (Task_H, Task_M, Task_L) med

hög (3), mellan (2) och låg (1) prioritet. Det högprioriterade och lågprioriterade tasket har en gemensam resurs.



Figur 4.6: Delad resurs

Task_L börjar sin exekvering och tar efter ett tag resursen. Sedan blir Task_H och Task_M ready och eftersom Task_H har högst prioritet avbryter det Task_L och påbörjar sin exekvering. När det vill ta den gemensamma resursen blir det blockerat eftersom resursen redan är tagen och Task_H hamnar i tillståndet waiting.

Task_M är nu högst prioriterat och får därför exekvera tills det har exekverat klart. Nu får Task_L exekvera tills det har släppt sin resurs och först då kan Task_H fortsätta sin exekvering. Det betyder att Task_H i princip har ärvt Task_L's prioritet eftersom alla task med högre prioritet än Task_L fick köra innan Task_H.

4.4.2 Priority Ceiling Protocol (PCP)

Priority ceiling protokollet används för att undvika de problem som kan uppstå när lås används för att förhindra samtidig access. Enligt protokollet tilldelas alla resurser en statisk takprioritet som är minst den högsta av prioriteterna på de task som har access till resursen. Prioriteten ska dock vara lägre än den lägsta bland de task som inte får ta den.

När ett task tar en resurs höjer det sin prioritet till resursens takprior-

itet och när det sedan släpper resursen återfår det sin tidigare prioritet. Trots att ett task har en statiskt tilldelad prioritet kan dess prioritet följdaktligen ibland ses som högre på grund av att det har tagit en resurs.

Exempel: Tre task har prioriteter och resurser enligt tabell 4.2

Task	**Prioritet**	**Resurser**
TASK_H	3	Resurs_1
TASK_M	2	Resurs_1 Resurs_2
TASK_L	1	Resurs_1 Resurs_2 Resurs_3

Tabell 4.2: Task-prioritet och resurser.

Takprioriteten för resurserna blir enligt PCP:

Resurs	**Takprioritet**
Resurs_1	3
Resurs_2	2
Resurs_3	1

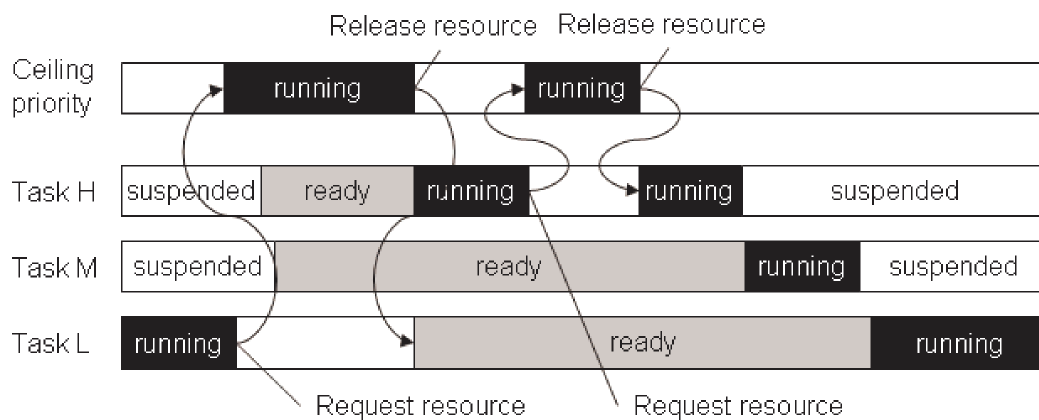
Tabell 4.4: Resursernas takprioriteter.

Ett task får endast ta resurser med lika eller högre prioritet än sig självt. Det betyder att endast Task_L kan ta Resurs_3 eftersom Resurs_3's prioritet är lägre än Task_M och Task_H's prioritet. Resurs_2 kan tas av Task_L och

Task_M och alla task kan ta Resurs_1.

4.4.3 PCP som lösning på prioritetsinversion

Om PCP tillämpas på det förgående exemplet ska alla resurser tilldelas en takprioritet. Task_H är det task som har högst prioritet av de task som delar på resursen och takprioriteten blir 3.



Figur 4.7: Delad resurs med PCP

På samma sätt som tidigare börjar Task_L exekvera och tar resursen. Task_L höjer sin prioritet till takprioriteten 3 och fortsätter sedan sin exekvering på den prioritetsnivån. Task_M och Task_H blir ready men eftersom Task_L nu exekverar på prioritetsnivån 3 har det fortfarande högst prioritet och får därför fortsätta sin exekvering.

Task_L exekverar tills det släpper resursen och därmed sänker sin prioritet till 1 igen. Då får Task_H köra som tack vare PCP endast har blivit blockerat av Task_L under tiden som Task_L håller den gemensamma resursen. När OSEK's resurser används för att synkronisera åtkomsten av delade minnesareor och liknande, går det aldrig att helt undvika att ett högprioriterat task blir blockerat av ett lågprioriterat. Med PCP minimeras i alla fall blockeringstiden samtidigt som deadlock och prioritetsinversion undviks.

4.4.4 Interna resurser

Ett task kan maximalt ha en intern resurs och det kan dela den med andra task men endast ett task åt gången har åtkomst till den interna resursen. När ett task övergår till tillståndet running tar det automatisk sin interna resurs, om det har någon, och höjer också sin prioritet till takprioriteten för den interna resursen. På samma sätt släpps resursen automatiskt när tasket lämnar tillståndet running.

Interna resurser beter sig precis som vanliga resurser förutom att det inte går att komma åt dem via OSEK's funktionsanrop. De används i enlighet med PCP men deras funktionalitet är strikt intern.

Detta gör att interna resurser kan användas för att skapa non preemtive task inom en viss grupp av task. Den interna resursens ges en takprioritet som är lika hög som den högsta prioriteten bland tasken i gruppen. När ett task som ska vara non preemtive inom gruppen får börja köra tar det automatiskt den interna resursen och höjer sin prioritet. Eftersom det nu har lika hög prioritet som det högst prioriterade av tasken i gruppen kan inget annat task i gruppen avbryta det. Andra högre prioriterade task kan fortfarande avbryta och få påbörja sin exekvering. [14]

4.4.5 Exempel

```
TASK basic_preemt_1
{
    PRIORITY = 5;
    RESOURCE = standard_resource;
};

RESOURCE standard_resource
{
    RESOURCEPROPERTY = STANDARD;
};
```

Figur 4.8: OIL-definition för en resurs och två task

I ett system finns flera både hög- och lågprioriterade task och i exemplet ovan visas bara koden för ett av tasken. Det task som är högst prioriterat och som kan ta standard_resource har prioritet 30 vilket med andra ord även blir exempelresursens takprioritet.

```
TASK(basic_preemt_1)
{
      int message;
      for (;;)
      {
            message = recive_message();
            status = getResource(standard_resource);
            if (status == E_OK)
            {
                  /* Print message to consol */
            }
            ReleaseResource(standard_resource);
      }
}
```

Figur 4.9: Exempelapplikation för resurser

Task basic_preemt_l vill när den mottar ett meddelande skriva ut det meddelandet på en konsoll utan att något annat högre task avbryter och påbörjar sin skrivning till konsollen. Någon form av lås behövs med andra ord för att förhindra samtidig åtkomst av konsollen. Detta åstadkoms genom att alla task som vill skriva först tar standard_resource och sedan när de har skrivit färdigt släpper de resursen igen och nästa task har då möjlghet att skriva till konsollen.

I exempelkoden ovan kommer alltså task basic_preemt_l att motta ett meddelande och sedan ta standard_resource och därmed höja sin prioritet till 30 som var resursens takprioritet. Inget annat task i systemet som vill kunna skriva till konsollen har en högre prioritet och basic_preemt_l kommer att få skriva hela sitt meddelande innan det släpper resursen.

4.5 Event

OSEK tillhandahåller medel för utveckling av händelsedrivna system med hjälp av funktionerna Wait-, Set- och ClearEvent. Task som i OIL-filen blir tilldelade event är av extended typ och kan synkroniseras med hjälp av dessa event. Event är inte oberoende objekt, utan måste vara tilldelade till ett task och har endast ett standardattribut, en eventmask.

Både ISR2, basic och extended task kan sätta ett event, det vill säga sätta en bit i en eventmask, och även kontrollera vilka event som är satta för ett specifikt task. Task kan dock bara vänta på och rensa sina egna event, vilket betyder att endast extended task kan vänta på ett event och på så vis gå över till tillståndet waiting. [14]

4.5.1 Exempel

```
TASK extended_preemt_h
{
   AUTOSTART = TRUE
   {
      APPMODE = OSDEFAULTAPPMODE;
   };

   EVENT = event_1;
   EVENT = event_2;
   PRIORITY = 30;
};

TASK basic_preemt_l
{
   PRIORITY = 1;
};

EVENT event_1
{
   MASK = 0x01;
};

EVENT event_2
{
   MASK = 0x02;
};
```

Figur 4.10: OIL-definition för två event och två task

I exemplet finns det två taskobjekt definierade med hjälp av OIL och de har prioriteterna 30 och 1. I objektet för task extended_preemt_h definieras även vilka två event det äger.

```
TASK(extended_preemt_h)
{
      StatusType status = 0;
      EventMaskType event_mask = 0;

      ActivateTask(baic_preemt_l);

      for (;;)
      {
            WaitEvent(event_1 | event_2);
            status = GetEvent(extended_preemt_h,
                              &event_mask);

            if (status == E_OK &&
                (event_mask & event_1))
            {
                  ClearEvent(event_1);
                  // Exekvera kod
            }

            if (status == E_OK &&
                (event_mask & event_2))
            {
                  ClearEvent(event_2);
                  // Exekvera kod
            }
      }
}

TASK(basic_preemt_l)
{
      for (;;)
      {
            // Exekvera kod
            SetEvent(extended_preemt_h, event_1);
            // Exekvera kod
            SetEvent(extended_preemt_h, event_2);
            // Exekvera kod
            SetEvent(extended_preemt_h,
                     event_1 | event_2);
      }
}
```

Figur 4.11: Exempelapplikation för event

Exempelkoden visar hur extended_preemt_h väntar på att event_1 eller event_2 ska sättas. När basic_preemt_l har satt ett eller flera av de event som task extended_preemt_h väntar på kontrolleras vilket eller vilka event som är satta. GetEvent ger en event_mask som innehåller alla satta event för ett task. Masken jämförs sedan mot respektive event och det eller de event som jämförelsen är sann för, rensas explicit med hjälp av ClearEvent.

4.6 Alarm och Counter

OSEK tillhandahåller alarm och counters som hjälpmedel för att hantera återkommande händelser. Alarm, är som namnet antyder, något som larmar efter en viss tid. Det kan även precis som för alarmklockor sättas att larma varje dag eller snarare sättas att larma varje gång en viss tidsrymd har passerat. Sådana alarm kallas för cykliska alarm och det är användaren av OSEK's API som bestämmer om ett alarm ska vara cykliskt eller inte.

Vad ett alarm ska utföra när det går ut är statiskt definierat i motsvarande OIL-objekt för alarmet. Handlingen som utförs är antingen att ett task aktiveras, ett event för ett specifikt task sätts, ett anrop till IncrementCounter sker eller att en alarm-callback-rutin körs.

För att ett alarm ska kunna gå ut vid en viss tidpunkt måste någon form av tidmätning finnas och alla alarm måste därför vara kopplade till en counter. En counter kan ha flera alarm kopplade till sig och vilka dessa är definieras statiskt.

Counter-tid mäts i ticks och alarm går ut antingen när de når ett visst tickvärde eller när ett visst antal ticks har gått. Med SetAbsAlarm sätts det absoluta tickvärde countern, som alarmet är kopplad till, ska uppnå för att alarmet ska gå ut. SetRelAlarm sätter istället ett relativt antal ticks innan alarmet går ut, det vill säga med så många ticks ska alarmets counter räknas upp innan det aktiveras.

I OSEK kopplas en counter till en systemklocka och dess tickvärde räknas upp med ett efter att systemklockan har slagit ett visst antal gånger.

Relationen mellan counterticken och systemklockan definieras i konfigurationsfilen.

I AUTOSAR har OSEK standarden utökats med funktionalitet för att öka en counters tickvärde med ett steg genom ett funktionsanrop. Funktionen IncrementCounter räknar endast upp en counter som är inte är kopplad till en systemklocka. [14]

4.6.1 Exempel

```
COUNTER hardware_counter
{
    MAXALLOWEDVALUE = 200;
    TICKSPERBASE = 3;
    MINCYCLE = 25;
    TYPE = HARDWARE;
};

COUNTER software_counter
  {
    MAXALLOWEDVALUE = 100;
    TICKSPERBASE = 1;
    MINCYCLE = 4;
    TYPE = SOFTWARE;
  };

ALARM alarm_setevent
{
    COUNTER = software_counter;
    ACTION = SETEVENT
    {
       EVENT = event_1;
       TASK = extended_preempt_h;
    };
};

ALARM alarm_incrementcounter
{
   COUNTER = hardware_counter;
   ACTION = INCREMENTCOUNTER
   {
      COUNTER = software_counter;
   };
};
```

Figur 4.12: OIL-definition för två alarm och två counters

I exemplet ingår även task extended_preemt_h som äger event_1 enligt exempel 4.5.1.

Counter hardware_counter har MAXALLOWEDVALUE 200, vilket betyder att den kommer att räkna upp till 200 innan den börjar om på noll. TICKSPERBASE 3 betyder att den bara kommer att räknas upp var tredje gång som systemklockan tickar. Minsta cykelvärde som alarm som är kopplade till countern kan ha är enligt MINCYCLE lika med 25 ticks.

Software_counter räknas upp med ett varje gång alarmet alarm_incrementcounter, som är kopplat till hardware_counter, går ut.

Alarm alarm_setevent är kopplat till software_counter och dess uppgift när det går ut är att sätta ett event för task extended_preemt_h.

```
TASK(extended_preempt_h)
{
     SetAbsAlarm(alarm_incrementcounter, 100, 100)
     SetRelAlarm(alarm_setevent, 4, 6)

     for (;;)
     {
          WaitEvent(event_1);
          ClearEvent(event_1);
          // Exekvera kod
     }
}
```

Figur 4.13: Exempelapplikation för alarm och counters

I extended_preemt_h sätts först alarm_incrementcounter till att vara ett absolut cykliskt alarm som slår då hardware_counter har tick-värdet 100. Countern börjar om från noll vart tvåhundrade tick eftersom dess maxvärde är 200.

Task extended_preemt_h sätter även ett relativt cykliskt alarm som slår

första gången efter 4 ticks och därefter vart sjätte tick. När det slår sätter det event_1 och extended_preemt_h kan då bli först ready och sedan running igen efter att ha väntat på just event_1.

Sammanfattningsvis kan det konstateras att extended_preemt_h kommer att få exekvera koden i sin loop vart 3600:de system tick.

Alarm alarm_incrementcounter slår vart tvåhundrade hardware_conter-tick vilket är vart tvåhundrade systemklocketick gånger 3, eftersom TICKSPERBASE = 3.

Alarm_setevent har ett cykelvärde på 6 tick och kommer därmed slå var sjätte gång alarm_incrementcounter slår, det vill säga var 3600:e gång som systemklockan tickar.

4.7 Felhantering

Alla API-anrop returnerar en av åtta fördefinierade statuskoder. Till exempel returneras E_OS_ID om argumentet som specificerar vilket task som ActivateTask ska aktivera är felaktigt. Går allt bra returnerar alla API-funktioner statuskoden E_OK förutom TerminateTask som inte returnerar alls om inget fel har upptäckts.

I stycke 4.8 beskrivs hur hookrutiner kan användas som hjälpmedel vid debuggning och felhantering.

4.7.1 Standard / Utökad

OSEK OS finns i standard- och utökad version. Den utökade versionen innehåller många felkontroller som underlättar i utvecklings- och testfasen av ett system. I tidskritiska system kan sedan onödig overhead, som tillkommer vid felkontroller, skalas bort genom att standardversionen av OSEK väljs. Standardversionen innehåller ett minimum av felkontroller eftersom systemet redan ska vara så pass testat att vissa felkontroller är överflödiga. [14]

4.8 Hook-rutiner

OSEK OS tillhandhåller ett antal hookrutiner vars gränssnitt är standardiserat men vars funktionalitet är användardefinierat eftersom de implementeras av användaren. Endast ett fåtal av OSEK's API funktioner kan användas från hookrutinerna. Hookrutiner anropas av operativsystemet och kan endast avbrytas av interrupt av kategori 1. Vilka hookrutiner som ska ingå i en applikation konfigureras via OIL.

StartupHook anropas från StartupOS efter att operativsystemet har initierats men före att det första tasket får processortid. Hookrutinen ger användaren en möjlighet till implementationsspecifik initiering.

ShutdownHook anropas från ShutdownOS och ger användaren en möjlighet att till exempel hantera sådant som att stänga ner I/O-komponenter innan systemet avslutas.

PreTaskHook anropas precis efter att ett task för första gången hamnar i tillståndet running och PostTaskHook anropas precis innan det lämnar running. OSEK ger med dessa hookar användaren ett hjälpmedel vid felsökning av systemet.

ErrorHook anropas om någon av OSEK's API-funktioner returnerar en felkod. Det är förbjudet med nestade ErrorHook-anrop så om ett fel sker i ett anrop till en API-funktion från ErrorHook, anropas inte ErrorHook igen. [14]

4.9 Application mode

OSEK kan köras i olika application modes som är anpassade för olika ändamål, till exempel en mode för fabrikstest och en annan för normal gång. I OSEK OS finns det krav att det måste finnas minst en application mode.

Varje application mode har sitt eget subset av objekt men det kan även vara så att till exempel ett task används i flera application modes om det

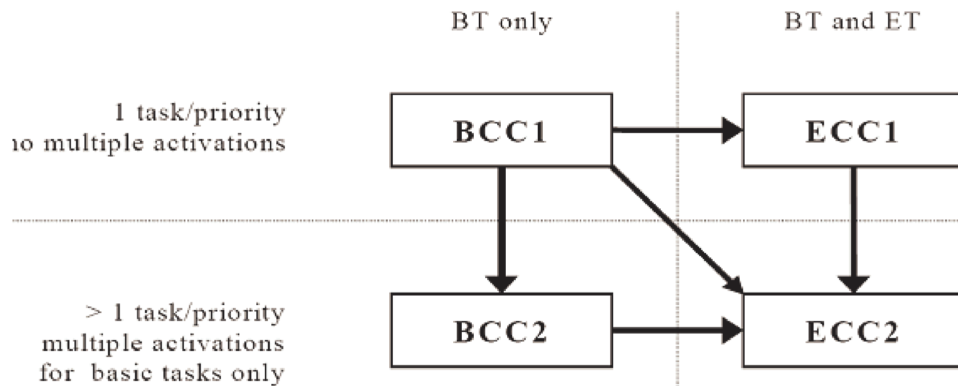
har samma funktionalitet. Som exempel kan man tänka sig att vid felsökning och utveckling kanske några extra task behövs som sedan är överflödiga i ett färdigtestat system. Application modes ger då utvecklaren ett smidigt sätt kunna byta mellan debuggläge och vanligt läge.

Vilken application mode som används bestäms innan uppstart och går inte att byta under körning. [14]

4.10 Conformance Classes

Det är meningen att OSEK OS ska kunna användas både i mikroprocessorer med låg prestanda och i mer komplexa kontrollenheter. För att ett operativsystem ska få kallas för OSEK-kompatibelt måste det först certifieras. För att möjliggöra certifiering på en delmängd av OSEK har fyra olika conformance classes definierats och dessa innehåller lite olika grad av funktionalitet för att passa ett så stort omfång av system som möjligt.

Vilken eller vilka conformance classes som är uppfyllda bestäms av om implementationen klarar av multipla task aktiveringar, det vill säga att flera instanser av samma task kan köras samtidigt, vilka tasktyper det finns och antalet task per prioritetsnivå. Det finns också minimikrav på hur många task, alarm, event, resurser och interna resurser som operativsystemet ska kunna hantera.



Figur 4.14: Conformance Classes

Bilden visar de fyra conformance klasserna BCC1, BCC2, ECC1 och ECC2. Klasserna är kompatibla i pilriktningarna. Mellan BCC[1] och ECC[2] skiljer det att ECC hanterar inte bara basic task utan även extended task. Mellan 1 och 2 skiljer det att för 1:orna behöver implementationen bara klara av ett task per prioritet och den behöver inte klara av multipla task aktiveringar vilket 2:orna ska uppfylla. [14]

[1] Basic Conformance Class
[2] Extended Conformance Class

Kapitel 5

OSE RTOS

Enea har utvecklat realtidsoperativsystemet OSE (Operating System Embedded) som är optimerat för inbyggda system. OSE används idag i komplexa distribuerade system världen över, framförallt inom telekommunikation. Operativsystemet är moduluppbyggt och funktionalitet kan läggas till eller tas bort beroende på vad som krävs för en viss tillämpning.

Det finns tre varianter av operativsystemet som är anpassade efter olika hårda realtids- och minnesanvändnings krav. OSE Delta, den största varianten av OSE, används till komplexa distribuerade system med mycket stora krav på pålitlighet.

OSE$_{ck}$ (OSE Compact Kernel) är anpassad för digitala signalprocessorer (DSP) och används i system med hårda minnes- och realtidskrav. Det är OSE$_{ck}$ som i examensarbetet har anpassats till OSEK och är därför den variant av OSE som kommer att beskrivas utförligare.

Det finns också ett mini OSE, kallat OSE Epsilon, som är framtaget i assembler och är den variant av OSE som har hårdast krav på låg minnesanvändning.

För att ge en känsla för vilka storlekar det rör sig om när man säger

liten, mellan och stor variant av OSE kan det nämnas att OSE Epsilon är ungefär 6 kB stort och OSE_{ck} är mellan 8 och 10 kB stort. OSE_{ck} går att göra mindre men vid den nämnda storleken ingår en hel del funktionalitet. OSE Delta är betydligt större, bara kärnan tar 100 kB minne. Vid 1-2 MB är OSE ett komplett operativsystem med till exempel IP-stack och filsystem.[10]

Kapitel 6

OSE_{ck}

OSE_{ck} är, som tidigare har nämnts, den version av OSE RTOS som är optimerad för DSP:er. Operativsystemet är framtaget för att användas både på system med endast en processor och på system som är distribuerade över flera processorer. En OSE applikation designas på samma sätt oavsett hur många processorer den körs på.

OSE_{ck} är helt event drivet och uppfyller kravet på fullständig determinism. Kärnan är också helt preemtive vilket betyder att interrupt kan ske när som helst, till och med under systemanrop. Även interrupt kan i sin tur bli preemtade av andra högre prioriterade interrupt. [10]

6.1 Konfiguration av OSE_{ck}

Konfigurering av OSE_{ck} görs med hjälp av Mkconfig. Programmet tar en konfigurationsfil med filsuffixet .con och producerar utifrån denna en C-fil. Den resulterande C-filen ska sedan länkas till applikationen. Nedan visas några exempelrader från en konfigurationsfil:

```
PRI_PROC(0, master, master, 128, 2)
PRI_PROC(0, slave, slave, 128, 10)
```

```
OS_INT(0, interrupt1, interrupt1, 19, 5)
START_HANDLER1(configure_cpu)
START_HANDLER2(task_initialization)
START_HANDLER2(StartupHook)
```

`PRI_PROC` och `OS_INT` används för att deklarera prioriterade processer respektive interrupt processer, dessa kommer förklaras senare i rapporten. De tar argumenten:

```
PRI_PROC(pid, processname, entrypoint, stacksize, priority)
OS_INT(pid, processname, entrypoint, interrupt-number, priori-
ty)
```

Där id 0 betyder att ett lämpligt id automatiskt väljs.

Starthandlers körs i den ordning de är deklarerade. [11]

6.2 Initiering

I OSE_{ck} finns det två typer av starthandlers som kan användas för initiering, `START_HANDLER1` och `START_HANDLER2`, och det kan finnas flera instanser av varje. De tar funktioner som argument och dessa körs i den ordning starthandlerna är deklarerade i OSE_{ck}'s konfigurationsfil.

Starthandlers av typen 1 anropas innan någon OS-initiering har skett och starthandlers av typen 2 anropas innan den första processen börjar köra. [11]

6.3 Processer

De enda enheterna i OSEck som blir exekverade av processorn är prioriterade processer och interrupt processer. Processer kan i OSE_{ck} vara antingen statiska eller dynamiska. Statiska processer skapas vid uppstart, via OSE_{ck}:s konfigurationsfil, och existerar ända tills programmet avslutas.

Dynamiska processer kan skapas och termineras under körning.

Den vanligaste typen av processer är prioriterade processer som implementeras som en oändlig loop. Dessa kan bli avbrutna av interrupt eller högre prioriterade processer men fortsätter sedan när det blir deras tur att köra igen.

I OSE_{ck} finns det både mjukvaruinterrupt och hårdvaruinterrupt. Eftersom mjukvaruinterrupten inte behöver kommunicera med kärnan, har de en lägre responstid än hårdvaruinterrupten. En interruptprocess kör från början till slut såvida den inte blir avbruten av ett annat interrupt.

6.3.1 Prioriteter

Alla processer ska ha en prioritet mellan 0 och 31 och det kan finnas fler än en process per prioritet. Ju lägre siffra desto högre prioritet, det vill säga 0 utgör högsta möjliga prioritet. Alla processer är ordnade efter samma prioritetsskala men interrupt har alltid högre prioritet än prioriterade processer.

6.3.2 Tillstånd

OSE_{ck}'s processer kan hamna i tre olika tillstånd, running, ready och waiting.

När en process får tillgång till processorn hamnar den i tillståndet running. Det kan därmed endast finnas en process åt gången, och per processor, som är running.

Alla processer som vill få processortid är placerade i en ready-kö där den process med högst prioritet står på tur att bli running. Det finns även en kö per prioritetsnivå som tillämpar FIFO[1]principen. Om en process blir preemtad blir den dock placerad först i kön för sin prioritetsnivå.

[1]First In First Out

En process är i waiting-tillståndet om något av de följande påståendena är sant:

- processen väntar på en signal[2]
- processen väntar på att ett delay-anrop[3] ska köras klart
- processen väntar på en semaphor
- processen har blivit stoppad

[11]

6.4 Schemaläggare

Schemaläggaren låter den process som har högst prioritet köra. Finns det flera processer med samma prioritet är det den som har blivit avbruten som får köra först, alternativt den som har väntat längst på att få processortid. En process kan bli preemtad när som helst, även under ett systemanrop, av högre prioriterade processer eller interrupt. [11]

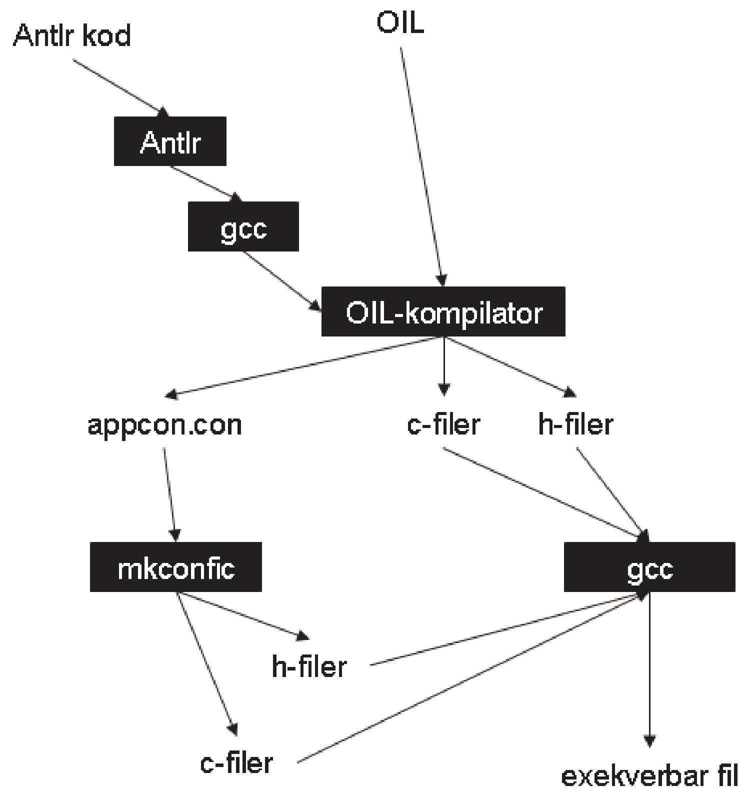
[2]Signaler används i OSE$_{ck}$ för att skicka meddelanden mellan processer och de kan innehålla data.

[3]Med ett delay-anrop släpper en process processorn under ett visst antal ticks.

Kapitel 7

OIL-tool

För att utifrån en OIL-applikation kunna skapa konfigurationskod, i språket C för OSEK och på mkconfig-format för OSE_{ck}, behövdes ett specialanpassat verktyg. I examensarbetet ingick det därför att utveckla verktyget OIL-tool som tillhandahåller den önskade funktionaliteten.



Figur 7.1: Från OIL-kod till konfigurationsfiler

När OSEK-lagret ovanpå OSE_{ck} implementerades skapades ett antal strukturer med information som inte OSE_{ck} tillhandahåller. För nästan alla OSEK-objekt finns det två strukturer med nödvändig information om objektet. Den ena strukturen innehåller konstant information som genereras vid uppstart och kan till exempel för ett alarmobjekt vara vilken counter alarmet är kopplat till och vad som ska hända när alarmet går ut. Den andra strukturen innehåller information som inte är konstant och som påverkas av OSEK's systemanrop, till exempel om ett alarm används eller ej, när det ska gå ut och vilket cykelvärde det har.

Såsom har beskrivits tidigare definieras OSEK-objekten i en eller flera OIL-filer. Hur informationen om objekten plockas ut och omvandlas till konfigurationskod beskrivs nedan.

7.1 Parsning

För att skapa en parser för OIL-filer har programmet ANTLR (ANother Tool for Language Recognition) använts. ANTLR är ett verktyg för att utifrån en grammatisk beskrivning skapa bland annat parsers och kompilatorer. I OIL specifikationen finns grammatiken för OIL specificerad på ett format som är mycket likt BNF (Backus-Naur Form) och med hjälp av den har en grammatikfil som ANTLR förstår, skapats. [1]

I första steget av parsningen genereras ett parse-träd utifrån den grammatik som beskriver OIL-syntaxen. Från parse-trädet skapas ett AST (Abstract Syntax Tree) som manipuleras bland annat genom eliminering av triviala tokens såsom operatorer och skiljetecken. Virtuella noder kan införas i trädet för att det ska bli tydligare och lättare att traversera. Noderna i trädet ordnas sedan efter vad som inbördes ska vara rötter, förälder- och barnnoder. Med hjälp av programmet ANTLR Works[1] kan ett träd visualiseras, vilket underlättar vid omskrivning av trädet.

Utifrån en ANTLR-grammatikfil genereras Lexer och Parser filer som innehåller information om AST-trädet och hjälpmedel för att traversera och även omforma trädet. Sedan återstår arbetet med att traversera trädet och plocka ut den information som behövs för att generera konfigurationsfilerna för OSEK och OSE_{ck}.

Innan en C-konfigurationsfil kan genereras måste det först kontrolleras att den OIL-fil som användaren har angivet är korrekt. Leverantören av systemet tillhandahåller en fil som innehåller OSEK's implementationsdefinition och som är en beskrivning av vad användarens OIL-fil kan och ska innehålla.

Några av de kontroller som genomförs är kontroll av att attributen har rätt typ, att endast befintliga objekt refereras, och att attributens värden är inom ett visst tillåtet intervall. Om ett fel upptäcks avslutas OIL-tool med ett felmeddelande som talar om på vilken rad felet uppstod.

[1] ANTLR Works är en integrerad utvecklingsmiljö för ANTLR

Om inget fel upptäcktes ska en C-fil med bland annat objektstrukturer skapas. Dessa ska även initieras med information om de olika objekten, såsom task- och resursprioritet. Det allokeras också utrymme som behövs under körning.

Kapitel 8

Implementation av OSEK lager

8.1 Task-anpassning

OSEK-task implementerades med hjälp av OSE_{ck}'s prioriterade processer. OSE_{ck}'s systemanrop för att starta och stoppa processer användes till OSEK-anropen ActivateTask, TerminateTask och ChainTask.

För att kunna göra de felkontroller som krävs av OSEK, behövdes dock lite extra information sparas i taskstrukturer. Ett task kan enligt OSEK aktiveras flera gånger vilket betyder att det blir ready igen direkt efter att det har terminerats. Det specificeras i OSEK's konfigurationsfil hur många gånger ett task får aktiveras. Information om max antal aktiveringar sparas i den konstanta taskstrukturen medan information om hur många gånger tasket har blivit aktiverat sparas i den ej konstanta strukturen.

OSE_{ck} tillhandahåller information om vilken process som kör och i vilket tillstånd en process befinner sig. Detta användes för att implementera OSEK-funktionerna GetTaskId och GetTaskState.

8.2 Resurs-anpassning

I OSEK's API ingår det två funktioner för att hantera resurser, GetResource och ReleaseResource. I dessa kontrolleras bland annat att det är ett task som anropar funktionerna och inte ett interrupt, att task endast tar resurser med högre eller lika prioritet som den statiska taskprioriteten och att resurser släpps i omvänd ordning som de togs.

I OSEK implementeras resurser med hjälp av PCP och protokollet ser till att deadlock och prioritetsinversion undviks. Det är också garanterat att användning av OSEK's resurser inte leder till att task hamnar i tillståndet waiting. I OSE$_{ck}$ kan semaphorer användas för att undvika samtidig åtkomst av delade resurser men användaren av dessa måste då själv se till att en semaphor endast kan användas av ett task i taget och själv sätta upp protokoll för att processer inte ska behöva vänta på semaphorer. Dessutom är det även upp till användaren att förhindra deadlock och prioritetsinversion. OSEK-resurser utgör med andra ord en mycket smidig lösning på problemet med delade resurser.

Nackdelen med OSEK's resurshantering är att alla task måste vara statiskt definierade för att takprioriteten på resurserna ska kunna räknas ut på ett smidigt sätt. I OSE$_{ck}$ går det att skapa nya processer dynamisk vilket kan användas för att köra flera instanser av samma kod och därmed spara systemresurser. Den funktionaliteten är dock inte tillåten i OSE$_{ck}$ med OSEK.

Eftersom OSEK använder sig av priority ceiling protokollet ska alla resurser tilldelades en takprioritet beroende på vilka task som vill använda resursen. Tack vare att det är specificerat i OIL-filen vilka resurser ett visst task är intresserat av, kan takprioriteten beräknas redan i OIL-tool och sedan sparas i en resursstruktur.

I ReleasResource kontrolleras det att resurser släpps i omvänd ordning som de togs och dessutom får ett task tillbaka prioriteten det hade innan det tog resursen. I en taskstruktur sparas en referens till den senast tagna resursen samt den statiska taskprioriteten, det vill säga den prioritet som specificeras i taskobjektet i konfigurationsfilen. Om ett task tar flera resurser

sparas de som en länkad lista, för varje ny resurs sparas en referens till den resurs som tasket tog senast och vilken prioritet ägartasket för tillfället har.

För att kontrollera att resurserna släpps i rätt ordning jämförs helt enkelt resursidentiteten på den resurs som ska släppas med den resursreferensidentitet som finns lagrad i taskstrukturen. Överensstämmer de inte returneras en felkod och OSEK's felhanterare anropas eventuellt. Annars släpps resursen genom att resursreferensen som finns lagrad i taskstrukturen sätts att peka på den resursreferens som sparats i resursstrukturen, det vill säga den resurs som nu kommer att vara den sist tagna. Sedan ändras taskprioriteten till den ägarprioriteten som finns lagrad i resursstrukturen som tillhör den nya sist tagna resursen.

8.2.1 Interna resurser

Interna resurser är implementerade precis som vanliga resurser med den skillnaden att de tas och släpps automatiskt vid schemaläggningspunkter i koden. De behöver bara tas när ett task blir running för första gången och när det lämnar tillståndet waiting. De släpps när ett task terminerar eller anropar WaitEvent. En intern resurs behöver alltså inte släppas när ett task blir preemtat eftersom task per definition bara kan bli avbrutna av task som inte använder samma resurs.

8.2.2 Schemaläggare som resurs

Enligt OSEK-specifikationen ska alltid en resurs, RES_SCHEDULER, tillhandahållas. Resursen används för att ta schemaläggaren. I praktiken betyder det helt enkelt att RES_SCHEDULER är en resurs med högsta möjliga prioritet, och att ett task inte kan bli avbrutet medan det håller resursen.

8.3 Alarm- och Counter-anpassning

En intern CounterCallback funktion krävdes för att alarm- och counterfunktionaliteten hos OSEK skulle kunna implementeras. OSE_{ck} tillhandahåller funktionalitet för att registrera en funktion så att den anropas vid

varje OSE_{ck} tick. Den interna CounterCallback funktionen registrerades och vid varje OSE_{ck}-tick räknas en intern tickparameter upp med ett.

Det är i OIL-objektet för varje counter definierat hur många av systemklockans tick som ska gå innan counterns tick ska räknas upp med ett. Alla counters räknas alltså inte upp samtidigt och inte heller vid varje OSE_{ck}-tick. Det är CounterCallback-funktionens uppgift att hålla reda på tickvärdet för varje counter och att räkna upp detta varje gång ett visst antal OSE_{ck}-ticks har gått. Den ska också kontrollera vid varje tick om något alarm har gått ut och om så är fallet utföra den handlig som alarmet anger. Innan handlingen utförs nollställs alarmet såvida det inte är ett cykliskt alarm.

8.4 Event-anpassning

Vid taskinitieringen tilldelas alla task varsin semaphor som initieras till noll. När OSEK-funktionen WaitEvent anropas för ett visst task, sätts först dess semaphor till 0 igen för att eventuella gamla signaler ska rensas. Sedan anropas OSE_{ck}'s systemanrop för att vänta på en semaphor och värdet på semaphoren räknas ner med ett. I SetEvent signaleras därefter semaphoren för ett visst task och semaphorvärdet räknas då upp med ett.

8.5 ISR-anpassning

Interruptrutinerna i OSEK är rakt mappade mot OSE_{ck}'s interruptprocesser. OSEK ger med sina två typer av interrupt en optimeringsmöjlighet men lämnar det upp till implementationen om denna möjlighet utnyttjas eller inte. För enkelhetens skull och eftersom det man vinner med optimeringen är minimalt, implementerades interrupt av kategori 1 och kategori 2 på samma sätt.

I OSEK finns en valfri del som beskriver hur även interrupts kan använda sig av resurser och utnyttja fördelarna med PCP. Om interrupts ska ges möjlighet att ta resurser, måste det gå att ändra deras prioritet. I OSE_{ck}

går det visserligen att ändra prioritet på intterupts i runtime men det är inget som rekommenderas. Därför togs beslutet att den valfria delen av OSEK, det vill säga att implementera PCP även för interrupts, inte skulle utföras. Se stycke 10.3 för ytterligare kommentarer.

8.6 Hook rutiner

Alla hookrutiner är användarimplementerade men en del arbete krävdes för att identifiera var i koden de ska anropas.

8.6.1 StartupHook

Den sista starthandlern av typ 2 tar OSEK's StartupHook som argument såvida det är definierat i OIL-filen att det finns en sådan. StartupHook körs alltså efter att all annan initiering av operativsystemet har skett men innan det första tasket får börja köra.

8.6.2 PreTaskHook och PostTaskHook

OSE_{ck} tillhandahåller en SWAP_HOOK som körs vid kontextbyte mellan tasks. Till att börja med utreddes det om den kunde användas för att implemenetera Pre- och PostTaskHook. Den ger dessvärre bara information om vilket task som kommer att börja köra, inte vilket som nyss körde. Man tänkte sig dock att information om vilket task som har kört kunde sparas i en global variabel. SWAP_HOOK körst dock vid varje kontextbyte, det vill säga även när interrupt körs. Även här är det möjligt att hålla reda på att användarens hookar inte ska anropas men det kräver lite extra overhead i form av kontroller av vilken typ av process som kör.

Enligt specifikationen ska OSEK's hookar endast anropas precis före att ett task startas första gången och precis efter att det avslutas. De ska alltså inte anropas när ett task bara blir preemtat vilket även detta är tillfällen när SWAP_HOOK kommer att köras. Ännu fler kontroller skulle alltså tillkomma för att kunna använda OSE_{ck}'s SWAP_HOOK och därmed orsaka en viss overhead.

Eftersom det första alternativet inte kändes helt bra utreddes det om det gick att identifiera alla sschemaläggningspunkter för task på ett smidigt sätt. Det visade sig att det inte alls var några problem och en mycket simplare lösning implementerades istället för den först påtänkta.

När schemaläggning sker och ett task aktiveras för första gången (eller aktiveras efter att ha blivit terminerat) anropas en initieringsfunktion som bland annat nollställer taskets semaphor. Det visade sig att för att stödja PreTaskHook funktionalitet behövdes enbart ett anrop till användarens def-inierade PreTaskHook-funktion göras i intieringsfunktionen för task.

Schemaläggningspunkten för PostTaskHook är precis efter att ett task har avslutas och istället för att använda OSE$_{ck}$'s SWAP_HOOK görs ett anrop till användarens PostTaskHook alldra sist i TerminateTask.

Lösningen som till slut implementerades krävde alltså ingen onödig overhead och inga globala variabler vilket, om möjligt, är bra att undvika.

8.6.3 ErrorHook

Från alla felkontroller som görs i OSEK funktionerna returneras en felkod om ett fel hittas. Innan den felkoden returneras, sker också ett anrop till en intern felhanteringsfunktion. Om det är definierat i OIL-filen att det finns en ErrorHook funktion sparas information om i vilket funktionsanrop ett fel uppstod, felkod och alla inparametrar till funktionen. Därefter sker ett anrop till ErrorHook, i vilken användaren sedan kan analysera den sparade informationen och vidta en eventuell felåtgärd.

8.7 Schemaläggare

OSE$_{ck}$'s schemaläggare är prioritetsbaserad precis som OSEK specifierar att den måste vara. Frågan var dock om den placerade preemtade tasks först eller sist i sin prioritetskö. Många schemaläggare är ”rättvisa” och låter hela tiden det task som har väntat längst på att få processortid,

och har högst prioritet, execvera. Genom att studera OSE_{ck}'s källkod och sätta upp några enkla testfall kunde det konstateras att task hamnade först i sin prioritetskö. Detta betydde att OSE_{ck}'s schemaläggningsprincip följde OSEK-standarden och därmed att inga ändringar i kärnan av OSE_{ck} behövde göras, vilket först hade befarats.

8.8 Conformance Classes

Alla obligatoriska delar av OSEK är implementerade, vilket betyder att OSE_{ck} med OSEK-lager uppfyller alla fyra conformance classes.

Kapitel 9

Prestanda

Ett av målen med examensarbetet var att undersöka hur OSEK-lagret påverkade operativsystemets prestanda. Både storleksmätningar på OSEK-lagret och tidmätningar på dess API och initiering har gjorts för att skapa en bild av vilken overhead lagret orsakar.

9.1 Förutsättningar

Alla prestandamätningar gjordes på processorn MPC5554 vid kristallfrekvensen 8 MHz. Endast internt RAM har använts och cache-funktionaliteten har varit avslagen under mätningarna. Anledningen till att ingen cache-funktionalitet har använts är för att mätningarna ska kunna upprepas och alltid ge samma resultat. Optimeringsnivån O2 användes för gcc kompilatorn eftersom den ger en balans mellan bra och säkra optimeringar. Det kan nämnas att det inte har ingått i examensarbetet att optimera koden för vare sig snabbhet eller storlek.

9.2 Jämförelse av storlek

I realtidsoperativsystem för inbyggda system är minnet ofta mycket begränsat och det är därför naturligtvis viktigt och intressant att veta hur

mycket mer minne som går åt på grund av OSEK-kompatibiliteten. Målet med mätningarna i det här stycket, är därför att ge en god bild av den extra overheaden och vad den i vissa fall beror på.

I tabellerna är storleken uppdelad i tre sektioner .text, .data och .bss. I .text delen ligger maskinkodsinstruktionerna som ska exekveras och för att det inte ska vara möjligt att skriva över dessa är .text delen oftast skrivskyddad. Datadelen är uppdelad i .data, som innehåller redan initierad data, och .bss som innehåller oinitierad data.

9.2.1 Storlek på OSEK-lager

Eftersom det är ganska svårt att plocka ut bara de delar som tillhör OSEK lagret har först en mätning av enbart OSE$_{ck}$'s storlek gjorts. OSE$_{ck}$'s storlek jämförs sedan med storleken av OSE$_{ck}$ med OSEK-bibliotek för att ge en bild av hur mycket overhead OSEK-bibliteket orsakar.

OSEK skapar overhead, även om inga API funktioner länkas med och därför finns det i tabellen angivet storleken för OSEK både med och utan API funktioner. Ju fler objekt som ingår desto större blir givetvis OSEK-lagret i storlek eftersom information om alla objekt sparas. I tabellen visas därför storlekar för OSEK utan objekt och OSEK med ett objekt av varje typ.

Beroende på om det är standard eller utökad versionen av OSEK, så görs olika många felkontroller och därför har overhead mätts både för standard och utökad version.

OS-Versioner	**.text**	**.data**	**.bss**
OSE_{ck}	59112	6552	55312
OSE_{ck} med OSEK, utan objekt och API funktioner	61432	6552	58712
OSE_{ck} med OSEK, standardversion, ett av varje objekt, hela API	66624	6576	59296
OSE_{ck} med OSEK, utökad version, ett av varje objekt, hela API	68048	6576	59296

Tabell 9.1: Storlek i kB på OSE_{ck} med och utan OSEK

OSE_{ck}-storleken, som anges är för en helt tom OSE_{ck}-applikation, utan några av OSE_{ck}'s systemanrop. Att OSE_{ck} ändå blir nästan 60 kB stort beror på bland annat C's standardbibliotek länkas med.

Mellan OSE_{ck} och OSE_{ck} med OSEK-lager för tomma applikationer skiljer det lite drygt 2kB, vilket betyder att OSEK-lagret alltid ger en liten overhead. Overheaden kommer sig bland annat av att där alltid måste finnas ett cpu-objekt och två resurser. Den ena resursen kräver OSEK-specifikationen medan den andra är en intern resurs med högsta prioritet som används för att göra ett task non preemtive. I stycke 10.3 tas det upp som ett förbättringsförslag att den senare resursen endast borde finnas om det finns non preemtiv tasks angivna i OIL-filen.

Givetvis kommer standardversionen av OSEK vara något mindre än den utökade versionen eftersom det ingår fler felkontroller i den utökade.

Slutsatsen av den här mätningen är att OSE_{ck} med ett OSEK-objekt av varje typ och hela OSEK's API endast blir 5.5 kB stort i sin standardversion och den utökade versionen 7 kB stort.

9.2.2 Storlek på objektfiler

Varje API-funktion är implementerad i en egen fil och storlekarna på alla objektfiler finns listade i tabellen nedan. Det finns även en fil per objekttyp, till exempel autosar_task.c och autosar_resource.c, som innehåller hjälpfunktioner för att till exempel göra felkontroller. Deras objektfiler finns också listade i tabellen.

Filnamn	**Standard / Utökad version**		
	.text	.data	.bss
autosar_task.o	656 / 656	0 / 0	0 / 0
autosar_activatetask.o	208 / 248	0 / 0	0 / 0
autosar_terminatetask.o	144 / 248	0 / 0	0 / 0
autosar_chaintask.o	104 / 104	0 / 0	0 / 0
autosar_schedule.o	84 / 156	0 / 0	0 / 0
autosar_gettaskid.o	64 / 64	0 / 0	0 / 0
autosar_gettaskstate.o	180 / 236	0 / 0	0 / 0
autosar_interrupt.o	312 / 312	0 /0	8 / 8
autosar_resource.o	624 / 624	0 / 0	0 / 0
autosar_getresource.o	144 / 260	0 / 0	0 / 0
autosar_releaseresource.o	96 / 196	0 / 0	0 / 0
autosar_setevent.o	92 / 280	0 / 0	0 / 0
autosar_getevent.o	64 / 162	0 / 0	0 / 0
autosar_clearevent.o	144 / 168	0 / 0	0 / 0
autosar_waitevent.o	144 / 300	0 / 0	0 / 0
autosar_alarm.o	288 / 288	0 / 0	0 / 0
autosar_getalarm.o	172 / 212	0 / 0	0 / 0
autosar_getalarmbase.o	80 / 128	0 / 0	0 / 0
autosar_setrelalarm.o	188 / 316	0 / 0	0 / 0
autosar_setabsalarm.o	148 / 272	0 / 0	0 / 0

autosar_cancelalarm.o	108 / 148	0 / 0	0 / 0
autosar_counter.o	820 / 820	0 / 0	4 / 4
autosar_incrementcounter.o	48 / 104	0 / 0	0 / 0
autosar_error.o	544 / 544	20 / 20	0 / 0
autosar_startos.o	32 /32	0 / 0	0 / 0
autosar_getactiveapplicationmode.o	8 / 8	0 / 0	0 / 0

Tabell 9.2: Filstorlekar i kB

För de flesta filerna skiljer storleken lite beroende på om mätningen är gjord för den utökade versionen eller standardversionen. Detta beror helt enkelt på att det i den utökade versionen av OSEK ingår fler felkontroller och därmed blir storleken på filerna större. De filer som inte skiljer sig mellan versionerna inehåller inga OSEK-funktioner eller endast sådana där inga versionsspecifika felkontroller görs.

Ingen storlek ovan skiljer sig så mycket från de andra att det är värt att tas upp men det kan i alla fall nämnas att det endast finns tre objektfiler som har .data eller .bss storlekar. I filen autosar_error.c initieras en array och det är därför som det finns en .data storlek angiven. De två andra filerna som har .bss storlekar innehåller variabeldeklarationer som är globala inom filen.

9.2.3 Description och Control Block listor

Såsom tidigare har nämnts finns det två listor per objekt-typ, undantaget är event som inte är oberoende objekt och inte heller innehåller någon information utöver sin identifierare. Varje OSEK-objekt tilldelas en struktur som innehåller konstant information om objektet och en som innehåller ej konstant information. Alla konstanta strukturer för en viss typ av objekt

sparas i en description block lista för den objekttypen, och de ej konstanta i en control block lista.

Listnamn	**konstant**	**storlek på en objektstruktur**
task_description_block	0	16
task_control_block	0	280
interrupt_description_block	0	8
interrupt_control_block	0	256
resource_description_block	16	8
resource_control_block	24	12
alarm_description_block	0	28
alarm_control_block	0	12
counter_description_block	0	16
counter_control_block	0	4

Tabell 9.3: Storlek i kB på listor med information om OSEK-objekten

Mätningarna visade på ett mycket tydligt samband mellan antal objekt och liststorlek, liststorlek är lika med storlek_på_ett_objekt gånger antal_objekt. I några fall tillkommer dock en konstant storlek som kunde läsas ut genom att mäta storleken på en helt tom lista. Hela formeln blir alltså: *liststorlek = konstant + storlek_ett_objekt × antal_objekt*.

Medlemmarna till strukturerna som listorna innehåller är till största delen heltal och strängar. Undantagen är dock control block listorna för task och intterupt. Anledningen till att dessa är så stora är att de innehåller en jmp_buf från C's standardbibliotek som håller undansparad data för att kunna starta om processen om den blir terminerad.

Endast resurslistorna innehåller data även om det inte finns några objekt

av den typen definierade. Detta beror på att resursen RES_SCHEDULER alltid ska tillhandahållas, se stycke 8.2.2. Dessutom innehåller de en intern resurs med högsta möjliga prioritet som används för att göra task non preemtive. Konstantens storlek är också just storleken på två resursobjektstrukturer.

9.3 Tidmätningar

I realtidsoperativsystem är det ofta väsentligt att veta hur lång uppstarttiden för systemet är och hur lång tid alla systemanrop tar. Därför har just sådana mätningar gjorts och resultatet av dessa redovisas i det här stycket.

Tidmätningarna har utförts på så vis att en funktion som startar tidmätningen har satts före det API-anrop som mätningen ska göras på, och ett annat som stoppar tidmätningen direkt efter. Därefter har en tredje funktion till uppgift att räkna ut differensen mellan start och stop, samt att dra ifrån den overhead som orsakas av tidmätningsanropen.

Tid mäts i cykler där en cykel är lika med 125 nano sekunder. Tidmätningar har gjorts utan cache för både standardversionen och den utökade versionen av OSEK's API.

9.3.1 Initiering

I realtidssystem är det viktigt att veta hur lång uppstarttid systemet har. Innan första tasket får börja köra, körs initieringsfunktioner för de objekt som ingår i systemet.

Initiering	**Initieringstid för X objekt** 1 cykel = 125 ns		
	1st	5 st	10 st
Initiering av task	8952	16456	25749
Initiering av resurser	316	668	1112
Initiering av alarm	130	394	926
Initiering av counters	268	524	848

Tabell 9.4: Initiering av objekt

Som tidigare har nämnts sparas information om alla objekt, förutom för events, i listor som innehåller objekt-strukturer. De listor som innehåller strukturer med konstant information initsieras i OSEK's konfigurationsfil. Det allokeras minne för listorna, som ska innehålla föränderlig information om objekten, i konifigurationsfilen men dessa behöver sedan initsieras vid uppstarten av systemet. Tabellen ovan visar hur lång tid initieringsfunktionerna för de olika objekten tar.

Eftersom initieringstiden är beroende av hur många objekt av en viss typ det finns, mättes tiden det tog att initiera 1, 5 och 10 objekt av en viss typ.

Alla task äger varsin semafor och dess värde initieras till 0 i initieringsfunktionen för task.

9.3.2 OSEK's API-anrop

Det är naturligtvis mycket intressant att veta hur lång tid ett API-anrop tar och i tabellen nedan visas resultatet av mätningarna på detta.

OSEK-systemfunktionalitet	**Antal cykler** (1 cykel = 125 ns) Standard / Utökad

ActivateTask (Preemt Basic Task)	963 / 1099
ActivateTask (Preemt Basic Task) Kontextbyte	4045 / 4203
ActivateTask (Preemt Extended Task)	963 / 1099
ActivateTask (Preemt Extended Task) Kontextbyte	4045 / 4203
ActivateTask (Non Preemt Basic Task)	963 / 1099
ActivateTask (Non Preemt Basic Task) Kontextbyte	4263 / 4427
TerminateTask (Preemt Basic Task)	1845 / 2113
TerminateTask (Preemt Basic Task) Kontextbyte	4793 / 5055
TerminateTask (Preemt Extended Task)	1845 / 2113
TerminateTask (Preemt Extended Task) Kontexbyte	4793 / 5055
TerminateTask (Non Preemt Basic Task)	2833 / 3089
TerminateTask (Non Preemt Basic Task) Kontexbyte	5781 / 6031
ChainTask (Preemt Basic Task)	4405 / 4545
ChainTask (Preemt Extended Task)	4405 / 4545
ChainTask (Non Preemt Basic Task)	4629 / 4769
Schedule	413 / 707
GetTaskID	307 / 313
GetTaskState	797 / 955
DisableAllInterrupts	49 / 49
EnableAllInterrupts	55 / 55
ResumeAllInterrupts	145 / 145
SuspendAllInterrupts	155 / 155
ResumeOSInterrupts	145 / 145
SuspendOSInterrupts	155 / 155

GetResource 1	813 / 1319
ReleaseResource 1	1483 / 1895
SetEvent	403 / 1647
ClearEvent	617 / 775
GetEvent	333 / 1603
WaitEvent	749 / 1235
GetAlarmbase	471 / 641
GetAlarm	811 / 951
SetRelAlarm	865 / 1287
SetAbsAlarm	721 / 1143
CancelAlarm	451 / 621
IncrementCounter	261 / 515
ErrorHook	241 / 241

Tabell 9.5: Tidmätning för OSEK's systemanrop

ActivateTask

I mätningarna på ActivateTask har anropet lagts i ett extended task och sedan undersöks hur lång tid det tar att aktivera respektive starta ett basic, extended eller non preemtive task.

Vid tidmätningen på AktivateTask mättes först hur lång tid anropet tar när det görs i ett högprioriterat task ,vilket betyder att ActivateTask endast kommer att aktivera ett task, inte starta det. Eftersom det inte är någon skillnad på att bara aktivera ett basic, extended eller non preemt task borde det inte heller ta olika lång tid att aktivera dessa. Reultatet av mätningarna blev som förväntat enligt tabellen ovan.

Sedan mättes även hur lång tid ActivateTask tar om ett task även startas, det vill att säga det är ett lågprioriterat task som aktiverar ett högprioriterat. Det betyder med andra ord att ett kontextbyte sker mellan tasken. Det intressanta med de senare mätningarna är att det tar olika lång tid att utföra ActivateTask beroende på vilket task som startats. Detta beror naturligtvis på att ett non preemtive task automatiskt ska ta den interna resursen som gör att tasket blir non preemtive, innan det faktiskt startar. Non preemtive task ska dock inte ta resursen om det bara aktiveras, vilket är skälet till att de första mätningarna blev lika för alla typer av task.

För basic och extended task blir mätningarna lika vid aktivering av tasken, men det tar givetvist längre tid att starta ett task jämfört med att bara aktivera det eftersom ett kontextbyte sker.

TerminateTask

Det har även för TerminateTask utförts mätningar på hur lång tid det tar att terminera ett task med och utan kontextbyte. Hur är detta då möjligt kan man fråga sig. Det går knappast att ha en instruktion efter TerminateTask anropet som stoppar och skriver ut resultatet av tidmätningen. Är ett task terminerat så är det. Tidmätningen för TerminateTask utan kontextbyte är därför gjord i själva koden för TerminateTask. Först och sist i koden sitter helt enkelt instruktioner som startar, stoppar och läser ut tiden.

Precis som för ActivateTask tar naturligtvis kontextbytet en stor del av tiden för att stoppa ett task. Kontextbyten är dyra och det visar verkligen siffrorna i tabellen på. Likaså tar det tid att släppa en intern resurs och det är även i det här fallet därför det tar längre tid att avsluta ett non preemtive task mot att avsluta de andra typerna av task. I ActivateTask tas den interna resursen först vid ett kontextbyte men eftersom det alltid blir ett kontextbyte vid ett anrop till TerminateTask kommer den interna resursen alltid att släppas. Det är därför som det tar längre tid att genomföra TerminateTask både med och utan kontextbyte.

Övrigt task API

ChainTask består av ett anrop till TerminateTask och sedan ett anrop till ActivateTask och tabellvärden för ChainTask visar på precis samma slutsatser som tidigare har dragits.

I övrigt kan det bara nämnas att mätningen på Schedule är gjord utan att anropet orsakade schemaläggning. Om så hade varit fallet hade tiden ökats med en liknande faktor som för Activate- och TerminateTask.

Interrupt API

I OSE_{ck} finns det bra funktionalitet för att slå av och på interrupt, vilket innebär att interrupt APIt var enkelt att implementera. Dessutom kräver inte interruptfunktionerna någon särskillt stor tidsåtgång tack vare dessa.

För interrupt APIt görs inga speciella felkontroller och det är också därför tiderna för standardversionen och den utökade versionen av OSEK är lika.

Resource API

I Get- och ReleaseResource måste OSE_{ck}'s systemanrop användas för att hämta ett tasks prioritet och för att eventuellt sätta om den prioriteten. Detta gör att dessa API-anrop tar förhållandevis lång tid att utföra.

När ett task släpper en resurs förändrar den eventuellt sin prioritet vilket betyder att OSE_{ck}'s schemaläggare kommer att kontrollera att ingen schemaläggning ska ske. Därför tar ReleaseResource längre tid att exekvera än GetResource som inte kan orsaka schemaläggning.

Event API

API anropet WaitEvent tar godtyckligt lång tid beroende på hur lång tid det tar innan det event, som det väntas på, sätts. Det kan däremot hända att overhead orsakas av att ett task börjar vänta på ett event som redan är satt och det är den tiden som anges i tabellen.

Alarm och Counter API

SetRelAlarm tar en något längre tid att utföra än SetAbsAlarm, vilket beror på att det relativa alarmvärdet först måste omvandlas till ett absolut värde och sedan kontrolleras att det är inom de tillåtna gränserna. Maxgränsen för vad ett alarm kan sättas till, beror på vilket maxvärde alarmets counter tillåter. För ett absolut alarm måste tiden som alarmet ska slå hamna inom ramen för vad en counter är satt att som max räkna upp till.

För ett relativt alarm adderas alarmtiden bara på det absoluta värde som alarmets counter står på och om det överskrider maxvärdet börjar countern om från början igen. Säg att en counter, för ett alarm, har maxvärdet 1000 och just nu står på 950. Om ett relativt alarm ska gå ut om 100 ticks, kommer det absoluta värdet som dess counter ska ha när alarmet går ut att vara 50.

ErrorHook

Om en felkontroll i ett API-anrop inte går igenom ska ErrorHook anropas och alla inparametrar till funktionen ska sparas undan. I mätningen anropas en helt tom ErrorHook och det är endast tiden för att spara undan parametrar och göra nödvändiga kontroller som visas i tabellen.

Kapitel 10

Diskussion

10.1 Krav på underliggande operativsystem

För att ett operativsystem ska kunna göras OSEK-kompatibelt måste det tillhandahålla en viss funktionalitet. Framförallt måste schemaläggningsprincipen vara av samma typ som OSEK specificerar, det vill säga prioritetsbaserad schemaläggning där FOIFO principen gäller för task av samma prioritet och där preemtade task hamnar först i sin prioritetskö. Att preemtade task hamnar först i prioritetskön för att bli execverade är ett krav för att OSEK's resurshantering ska kunna fungera och är därför ett krav för att anpassningen ska kunna göras.

Innan ett task körs måste det vara möjligt att execvera kod som till exempel har till uppgift att anropa PreTaskHook. Operativsystemet måste alltså tillhanda hålla möjligheten att köra kod innan ett task börjar exekveras genom att tillhanda hålla en hook eller möjlighet till att köra en initieringsfunktion. Detta eftersom OSEK kräver stöd för en PreTaskHook.

OSEK's task har visserligen en statisk prioritet men den kan ändras på grund av PCP. Detta betyder att det underliggande operativsystemet måste tillhandahålla stöd för att ändra processers prioritet under körning. Ska

PCP för interrupt kunna implementeras måste det dessutom vara möjligt att blanda interrupt och task prioriteter.

10.2 Ändringar i kärnan

Ett av målen med examensarbetet var att utreda om det var möjligt att skapa ett bibliotek ovanpå OSE_{ck} för att uppfylla alla kraven i OSEK. Om det visade sig att detta inte var möjligt skulle ändringar i OSE_{ck}'s kärna göras för att skapa OSEK kompatibilitet.

Det visade sig att OSE_{ck} erbjuder alla tjänster som behövs för att endast ett kompatibilitetsbiblotek ska vara nödvändigt för OSEK-kompatibilitet. Det behövdes därför inte göras några ändringar direkt i OSE_{ck} vilket var det resultat som Enea hoppades på.

Även om det var möjligt att helt undvika ändringar i kärnan, kunde fortfarande prestandamätningarna ha visat på att det fanns overhead som en integration med kärnan skulle kunna minska. Prestandamätningarna visade dock att ett OSE_{ck} med OSEK fortfarande är förhållandevis litet även när all API-funktionalitet ingår. Inte heller tidmätningarna visade på någon större extra overhead, alla värden hamnar inom rimliga gränser.

Att tidmätningarna blir så pass bra som de blir är egentligen inte förvånande. I OSE_{ck} idag görs det ingen skillnad på user och kernel mode eftersom OSE_{ck} inte har något minnesskydd. Att implementera ett sådant minnesskydd har två examensarbetare fått till uppgift att göra och det ska bli mycket intressant att se hur deras arbete kommer att påverka prestandan för operativsystemet.

10.3 Utvidgning och förbättringsförslag

Intentionen under arbetets gång har varit att underlätta och förbereda inför vidareutveckling av projektet. OSEK's kompatibilitetsbibliotek är bara första steget i att göra OSE_{ck} AUTOSAR-kompatibelt. Kompatibilitets-

biblioteket är dessutom bara en del i projektet, utöver det ska bland annat minnesskydd införas i OSE$_{ck}$.

Det finns även delar i kompatibilitetsbiblioteket som kan utökas eller förbättras. Ingen optimering av kod har gjorts och det finns sannolikt bättre lösningar på en del av de problem som har uppstått. Nedan punktas några förslag till utvidgningar och förbättringar.

- För att kunna återställa ett task, det vill säga, terminera och sedan starta ett task igen, lagras information i en jmp_buf. Bufferten tar mycket minne och det vore därför önskvärt att hitta en bättre lösning.
- En annan förbättring som kan göras, och troligen utan större besvär, är att införa stöd för flera application modes. OSEK kräver förnärvarande bara att ett application mode stöds och det är så det är implementerat nu.
- Alla obligatoriska krav i OSEK-specifikationen är uppfyllda men det valfria kravet som handlade om att även interrupt ska kunna ta resurser och använda sig av PCP är inte implementerat. I OSE$_{ck}$ är både interrupt och task indelade i en diskret prioritetsskala men interrupt har alltid högre prioritet än task. För att både task och interrupt ska kunna använda sig av PCP behöver deras prioriteter kunna blandas vilket det skulle krävas ändringar i OSE$_{ck}$'s kärna för att möjliggöra. Eftersom målet var att om möjligt undvika ändringa i kärnan, ignorerades det valfria kravet på att interrupt ska kunna ta resurser. Detta krav är dock fortfarande intressant och kanske är det värt att analysera ytterligare.
- I OIL-tool kontrolleras applikationsdefinitionen mot implementationsdefinitionen för att felaktiga referenser och typer ska kunna upptäckas. Nu verifieras dock inte implementationsdefinitionens korrekthet gentemot OIL-specifikationen vilket vore en bra utvidgning av OIL-tool för att kunna fånga upp ännu fler fel. Sådant som att implementationsdefinitionen endast innehåller de objekt som OSEK tillåter kontrolleras visserligen redan vid parsningen men det finns fler bra kontroller som kunde göras såsom att kontrollera att rätt attribut av

rätt typ finns med. Till exempel måste det finnas specificerat i implementationsdefinitionen att task ska ha en prioritet och att den är av typen integer.

- Non preemtive tasks görs non preemtive genom att det tar en intern resurs med högsta prioritet så fort de börjar köra. Den interna resursen finns alltid, oavsett om det finns några non preemtive tasks eller inte i applikationen vilket är onödigt. Utan allt för stort arbete bör problemet kunna korrigeras, så att den specifika interna resursen endast finns med om det finns något non preemtive task angivet i OIL-filen.

Kapitel 11

Andra OSEK kompatibla OS

Det finns ett ganska stort utbud av andra OS som på olika sätt är kompatibla med OSEK. Nedan följer en kort presentation av några OS från olika producenter med lite olika inriktning och omfattning.

EB [4], Elektrobit Automotive, är ett tyskt företag inom EB koncernen som tillhandahåller lösningar inom både mjuk- och hårdvara för bland annat bilindustrin. OSEK är en av flera standarder som man arbetar med inom EB tresos®familjen för lösningar till sina kunder.

RTA-OSEK [5], från den tyska företagsgruppen ETAS, är ett OS som passar för alla delar av bilindustrins ECU design. RTA-OSEK har implementerat AUTOSAR-OS V1.0 (SC-1) och OSEK/VDX OS V2.2.3 standarderna. Den kan tillhandahållas för mer än 20 vanliga microcontrollers.

CodeWarrior OSEKturbo [6] tillhandahålls av Freescale Semiconductor som är ett stort halvledarföretag inom Motorolakoncernen med säte i Texas, USA. Den är fullt OSEK/VDX kompatibel och designad för att ta minsta möjliga minnesutrymme, göra snabba omställningar och öka

återanvändbar-heten för den inbyggda applikationen.

MICRO C/OS-II(μC/OS-II) [12] är det amerikanska företaget Micriums huvudprodukt. MCIRO C/OS-II tillhandahåller bland annat en utvidgning som uppfyller OSEK/VDX standard.

AlphaOS [7] är ett OSEK kompatibelt RTOS framtaget av forskare på Zhejiang University i Kina.

PICos18 [13], från det franska företaget Pragmatec inc., är ett OS som baserar sig på OSEK/VDX. Det är ett open-source projekt för PICmicro microcontrollers från Microchip PIC18 familjen.

OpenOSEK [8] är även det ett fritt open-source projekt i commuityform som utvecklar ett RTOS som främst riktar sig mot säkerhetskritiska applikationer inom bilindustrin. OpenOSEK siktar på full överensstämmelse med OSEK/VDX.

Litteraturförteckning

[1] Antlr Parser Generator (2008) (Elektronisk).
Tillgänglig: http://antlr.org/ Hämtad: 2008-09-01.

[2] Autosar (2008). *Specification of Operating System v 3.0.2* (Elektronisk) Tillgänglig:
http://www.autosar.org/download/AUTOSAR_SWS_OS.pdf Hämtad: 2008-08-30

[3] Bremicker, Oliver et al (2004) *OSEK/VDX: OSEK implementation language, Specification 2.5* (Elektronisk) Tillgänglig: http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf Hämtad: 2008-08-30

[4] Elektrobit Corporation (2008) *EB* (Elektronisk) Tillgänglig:
http://www.elektrobit.com/static/en/index.html Hämtad: 2008-08-30

[5] ETAS (2008). *Etas - RTA-OSEK* (Elektronisk) Tillgänglig:
http://www.etas.com/en/products/992.php Hämtad: 2008-08-30

[6] Freescale Semiconductors (2008). CodeWarrior OSEKturbo Product Summery Page (Elektronisk) Tillgänglig:
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=OSEKTURBO Hämtad: 2008-08-30.

[7] Gu Yang et al (2005). "AlphaOS, an automotive RTOS based on OSEK/VDX: design and test"(Elektronisk). *2005 IEEE International*

Conference on Networking, Sensing and Control, p 174-9. Tillgänglig: IEEExplore. Hämtad: 2008-08-30.

[8] *OpenOSEK: home page* (2008) (Elektronisk) Tillgänglig: http://www.openosek.org/tikiwiki/tiki-index.php Hämtad: 2008-08-30

[9] OSEK/VDX portal (Elektronisk) Tillgänglig: http://www.osek-vdx.org/ Hämtad: 2008-08-30

[10] Enea (2006). OSEck: Compact Kernel for Real-Time DSP Embedded Systems (Elektroniskt). Kista: Enea. Tillgänglig: http://www.enea.com/EPiBrowser/Literature%20(pdf)/Pdf/Not%20 leadgenerating/Datasheets%20and%20Brochures/OSEck_Datasheet.pdf Hämtad: 2008-09-01

[11] OSEck Kernel User's Guide. OSEck 3.3 Version 2 (2007). Kista: Enea Software AB

[12] Micrium (2008). *MICRO C/OS-II RTOS OSEK Layer* (Elektronisk). Tillgänglig: http://www.micrium.com/products/rtos/kernel/osek.html Hämtad: 2008-08-30.

[13] Pragmatec inc. (2006) *PICOS18* (Elektronisk) Tillgänglig: http://www.picos18.com/index_us.htm. Hämtad: 2008-08-30

[14] Spohr, Jochem (2005) OSEK/VDX Operating System, Specification 2.2.3 (Elektronisk) Tillgänglig: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf Hämtad: 2008-08-30

Avdelning, institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

2008-10-29

Linköpings universitet

Språk
Language

- [X] Svenska/Swedish
- [ ] Engelska/English
- [ ] ___________

Rapporttyp
Report category

- [ ] Licentiatavhandling
- [X] Examensarbete
- [ ] C-uppsats
- [ ] D-uppsats
- [ ] Övrig rapport
- ___________

ISBN —

ISRN LIU-IDA/

Serietitel och serienummer
Title of series, numbering

ISSN —

URL för elektronisk version

Titel
Title

OSEK-kompatibilitet hos ENEA OSE_{ck}

Författare
Author

Jenny Palmberg
Lili Ren

Sammanfattning
Abstract

Målet med examensarbetet var att undersöka om det var möjligt att genom ett kompatibilitetsbibliotek se till att Eneas realtidsoperativsystem OSE*ck* kan uppfylla kraven i operativsystemsstandarden OSEK.

OSE*ck* visade sig tillhandahålla all efterfrågad funktionalitet och ett kompatibilitetsbibliotek som innehöll OSEK's API kunde därmed implementeras. Ett verktyg togs fram för att utifrån en fil, innehållandes objekt beskrivna i OSEK's konfigurationsspråk OIL, plocka ut den information som behövdes för att konfigurera både OSE*ck* och OSEK.

Slutsatsen av examensarbetet blev att det gick att göra OSE*ck* OSEK-kompatibelt genom ett yttre lager och att inga ändringar i OSE*ck*'s kärna var nödvändiga. Givetvis påverkar lagret operativsystemets prestanda negativt men det får ändå anses att dess prestanda fortfarande är så pass bra att en integration i OSE*ck*'s kärna ej behövs.

För att ett operativsystem ska kunna göras OSEK-kompatibelt måste det ha prioritetsbaserad schemaläggning samt att task som blir avbrutna hamnar först i sin prioritetskö. Dessutom måste det vara möjligt att exekvera kod precis innan ett task börjar köra för första gången eftersom det ska finnas stöd för en PreTaskHook.

Nyckelord
Keywords

Realtidsoperativsystem, OSEK