Faculteit Toegepaste Wetenschappen
Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout

Functies van Ware-tijdbesturingssystemen in Hardware

door

Philippe Faes

Promotor: Prof. dr. ir. D. Stroobandt
Scriptiebegeleiders:
Dr. ir. S. Goossens
Ing. J. De Ceuster en Ir. W. Meeus

Scriptie ingediend tot het behalen van de academische graad van
burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2002–2003

Voorwoord

Deze scriptie is in eerste plaats tot stand gekomen dankzij het Leuvense bedrijf CoWare, die het onderwerp hebben aangereikt. Zij hebben hun ontwerpspakket N2C ter beschikking gesteld en mij bijgestaan tijdens mijn stage in het bedrijf. Gedurende het ganse afstudeerproject bleven de mensen bij CoWare ter beschikking voor mijn technische vragen. In het bijzonder verdient Dr. Serge Goossens als begeleider van de scriptie mijn dank.

Ik wil ook mijn promotor Prof. Dirk Stroobandt en scriptiebegeleiders ing. Jan De Ceuster en ir. Wim Meeus bedanken. Ze hebben me steeds bijgestaan met technisch en strategisch advies.

Dr. Ronny Stobbaerts verdient eveneens vermeld te worden omdat hij mij bijna tien jaar geleden heeft doen kennismaken met de (hobby-)elektronica en hoogfrequenttechnieken, en zo mijn technologische interesse heeft aangescherpt.

Er zijn duizenden programmeurs die ervoor gezorgd hebben dat ik kon beschikken over een arsenaal kwalitatief zeer hoogstaande en vrije programma's om deze scriptie te maken. Ik hoop dat ik deze schuld voor een klein stuk kan terugbetalen door programmatekst bij te dragen aan het Dia project.

Ik ben mijn vriendin Leen zeer dankbaar omdat ze me heeft gesteund tijdens mijn werk aan deze scriptie en voor haar technisch en taalkundig advies. Tot slot zou ik mijn ouders willen bedanken voor hun jarenlange morele, financiele en logistieke steun. Mijn moeder heeft, ondanks haar eigen studielast en examens, tijd kunnen maken om mijn scriptie te proeflezen. Het is onnodig te zeggen dat deze scriptie en zelfs mijn ganse studie zonder hen beiden niet mogelijk zou geweest zijn.

Toelating tot bruikleen

Functies van Ware-tijdbesturingssystemen in Hardware

door Philippe Faes

Scriptie ingediend tot het behalen van de academische graad van
burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2002–2003

Promotor: Prof. dr. ir. D. Stroobandt
Scriptiebegeleiders: Dr. ir. S. Goossens, Ing. J. De Ceuster en Ir. W. Meeus

Universiteit Gent
Faculteit Toegepaste Wetenschappen
Vakgroep Elektronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout

Samenvatting

In deze scriptie wordt onderzocht welke de implicaties zijn van het implementeren van een *RTOS* in hardware. We noemen dit een *HWOS*.

We beschrijven de methodologie van *hardware-softwareco-ontwerp* voor ingebedde systemen en het gebruik van *CoWareC* en *SystemC*. Er wordt besproken wat de functie is van een klassiek RTOS, meer bepaald van de *scheduler* en *semaforen*. Deze functies kunnen ook in hardware worden uitgevoerd, wat mogelijk snelheidswinst zal opleveren en wat zeker meer flexibiliteit voor de systeemontwerper zal creëren.

Er zijn voor deze scriptie twee implementaties van een RTOS in hardware (HWOS) voorgesteld. De ene is verfijnd tot register-transferniveau, van de andere is enkel een gedragsbeschijving en een structuurbeschrijving gemaakt. Verder is er software in C geschreven om met het HWOS te kunnen werken. Ook werden een aantal routines in assembler code geprogrammeerd om taakwissels uit te voeren.

Het HWOS zal een module zijn waarbij de communicatie met de software afgebeeld wordt op geheugenadressen. Eén van de implementaties van het HWOS zal de processor kunnen onderbreken om taakwissels uit te voeren. Signalen van randapparaten kunnen opgevangen worden door het HWOS en zullen niet rechtstreeks de processor onderbreken. Op die manier neemt het HWOS de functie van de encoder voor onderbrekingsprioriteiten over.

We maken een schatting van de snelheid van het HWOS en concluderen dat er wel degelijk snelheidswinst te halen valt uit een HWOS.

Tot slot stellen we een aantal mogelijke toekomstige verbeteringen voor een HWOS voor. De complexiteit van de verbeteringen varieert van zeer eenvoudig te implementeren, tot misschien niet haalbaar.

Trefwoorden: RTOS, ware-tijdsystemen, hardware-softwareco-ontwerp, semafoor, scheduler

Inhoudsopgave

Lijst van figuren

Acroniemen

ADS ARM Developer Suite

AHB Advanced High-speed Bus

API Application Programmers Interface

ARM Advanced RISC Machines. ARM is een geregistreerd hanselsmerk van ARM Limited.

ASIC applicatiespecifiek geïntegreerd cicuit. Application Specific Intergrated Circuit

BCA Bus Cycle Accurate

CISC Complex Instruction Set Computer

CPU centrale verwerkingseenheid. Central Processor Unit

DDD Data Display Debugger

FIFO First-In-First-Out geheugen

FIQ snelle onderbrekingsaanvraag. Fast Interupt reQuest

FPGA herprogrammeerbare poortmatrix. Field Programmable Gate Array

GDB Gnu DeBugger

GPL GNU General Public License. Softwarelicentie die de vrijheid van de gebruiker garandeert

HWOS RTOS in hardware. HardWare Operating System

IC geïntegreerd circuit. Intergrated Circuit

IEEE Institute of Electrical and Electronics Engineers

I/O invoer en uitvoer. Input and Output

IPC interprocescommunicatie. Inter Proces Communication

IPE encoder voor onderbrekingsprioriteiten. Interrupt Priority Encoder

IRQ onderbrekingsaanvraag. Interrupt ReQuest

ISR onderbrekingsroutine. Interrupt Service Routine

LR terugspringregister. Link Register

N2C Napkin to Chip

NOP instructie zonder effect. No OPeration

OS besturingssysteem. Operating System

PC programmateller. Program Counter

POSIX Portable Operating System API

RAM lees-en-schijfgeheugen. Random Access Memory

RISC Reduced Instruction Set Computer

ROM enkel-lezengeheugen. Read Only Memory

RTC RTL-C

RTL register-transferniveau. Register Transfer Level

RTOS ware-tijdbesturingssysteem. Real-Time Operating System

SPARC Scalable Processor ARchiteCture

SP stapelwijzer. Stack Pointer

μC/OS-II MicroC/OS-II. [Lab02]

UML Unified Modeling Language. Symbolische taal voor systeemontwerp

UT tijdloos. UnTimed

VHDL Very high speed integrated ciruits Hardware Description Language

Lijst met Nederlandse woorden

In de computerwetenschappen worden vaak Engelse termen gebruikt in een Nederlandse tekst. We proberen in deze scriptie zoveel mogelijk de bestaande Nederlandse termen te gebruiken, of goede Nederlandse alternatieven voor Engelse termen voor te stellen. Om verwarring te vermijden bij taalgebruikers die enkel de Engelse termen gewend zijn, volgt hier een lijst van Nederlandse termen met hun Engelse vertaling. Verder is in programmateksten het Engels gebruikt en worden in de Nederlandse tekst de namen van variabelen en functies letterlijk overgenomen en dus niet vertaald.

barrière barrier

brontekst source text, source code

gegevensstructuur data structure

gegevenstype data type

hoofdingsbestand header file

hulpbron resource

hulpprogramma tool

knikker token

lijmlogica glue logic

onderbreking interrupt

onderbrekingsroutine interrupt service routine (ISR)

ontluizen debug

ontwerpprogramma development tool

opvulsel padding

overlopen overflow

postbus mailbox

programmateller program counter, PC register

randapparaat peripheral device

rij array

stapel stack

stapelwijzer stackpointer, SP register

stuurprogramma driver

taalwoord keyword

tijdslimiet deadline

uitgestelde sprong delayed branch

uitvoerbaar executable

wachtlijn queue

ware-tijdsbesturingssysteem real-time operating system (RTOS)

ware-tijdssysteem real-time system

webstek website

wedloop race

Hoofdstuk 1

Inleiding

Het doel van deze scriptie is te onderzoeken wat de mogelijkheden en moeilijkheden zijn bij het implementeren van functies van een ware-tijdbesturingssysteem (RTOS) in hardware.

In dit hoofdstuk wordt uitgelegd wat *hardware-softwareco-ontwerp* is en hoe dit leidt tot *interfacesynthese*. We bespreken daarna wat een RTOS is en waarom we dit willen implementeren in hardware.

1.1 Hardware-softwareco-ontwerp

1.1.1 Ontwerpmethodologie

Bij hardware-softwareco-ontwerp worden hardware en software op hetzelfde moment, in dezelfde ontwerptaal (bv. SystemC of CoWareC) en met dezelfde hulpprogramma's ontworpen. Momenteel is SystemC één van de belangrijke ontwerptalen voor co-ontwerp. CoWareC is een voorloper van SystemC.

De systeemontwerper maakt een gedragsbeschrijving op hoog niveau van alle componenten van het systeem. Dit kan bv. in CoWareC of SystemC. Deze beschrijving is *uitvoerbaar* en kan gebruikt worden om het gedrag van het systeem te valideren. Ze bevat echter geen informatie over de uiteindelijke implementatie van een component. Bijvoorbeeld een component die de grootste gemene deler berekent van twee getallen kan uiteindelijk geïmplementeerd worden als een algoritme dat uitgevoerd wordt op een processor. Een andere mogelijke implementatie is

een combinatorische schakeling die van twee 8-bitsgetallen de grootste gemene deler genereert.

Componenten in een co-ontwerp noemt men *modules*. Deze modules communiceren met elkaar via één of meerdere *poorten*. Men maakt onderscheid tussen poorten die de communicatie zelf initiëren (`master`-poorten) en poorten die wachten op signalen van andere poorten (`slave`-poorten). Bij de simulatie van de hoogniveaubeschrijving bestaat er geen tijdsnotie. In het CoWareC-jargon spreekt men van een simulatie op tijdloos (UT) niveau. Er bestaat ook een Bus Cycle Accurate (BCA) niveau, waar een systeemklok gedefinieerd moet zijn. De details van het communicatieprotocol dat gebruikt wordt, zijn niet gedefinieerd op dit UT niveau. Men zal van poorten lezen of naar poorten schrijven zoals men waarden van een variabele leest of erin schrijft.

Het is belangrijk dat het systeem al zeer vroeg gesimuleerd en gevalideerd kan worden. Dit levert snelheidswinst op omdat inconsistenties en fouten dan snel kunnen opgespoord en hersteld worden. Na het valideren van de hoogniveaubeschrijving kan men dit model stap voor stap verfijnen. Men moet beslissen welke modules in hardware zullen worden uitgevoerd en welke als software op de processorkern zullen draaien. De softwaremodules moeten niet aangepast worden. Elke simuleerbare specificatie (op UT-niveau) kan door Napkin to Chip (N2C) automatisch omgezet worden naar software voor de doelprocessor.

Voor de hardware-modules schrijft men een model op het BCA niveau. In dit simulatieniveau moeten de communicatieprotocollen van de poorten gedefinieerd worden. Een poort zal uit verschillende *signalen* bestaan en al naargelang welk protocol men gebruikt, signalen als `d`, `req`, `ack` en `en` (*data*, *request*, *acknowledge*, *enable*) bevatten. Voor informatie over de CoWare-protocollen verwijzen we naar [cwr02h]. Het gegevenstype van elke poort moet gedefinieerd worden. In SystemC is het mogelijk om tijdens de simulatie een poort met gegevenstype `int` (geheel getal) te definiëren zonder dat daarbij duidelijk moet zijn door hoeveel bits het geheel getal voorgesteld wordt. Wanneer er hardware beschreven wordt, zal men dit aantal bits precies moeten definiëren. Het BCA-model is tot op een klokcyclus nauwkeurig.

Het BCA-model wordt daarna verder verfijnd is tot op register-transferniveau (RTL). Vanaf dit niveau kan het automatisch worden vertaald naar Very high speed integrated ciruits Hardware Description Language (VHDL). Dit deel van CoWareC dat omgezet kan worden naar

VHDL (en dus naar hardware) noemt men RTL-C (RTC). Vanaf dat punt kunnen klassieke ontwerpprogramma's voor VHDL gebruikt worden.

1.1.2 Interfacesynthese

Zoals reeds aangehaald, kan N2C de UT-modellen omzetten in software voor de processorkern en de RTC-modellen (onrechtstreeks) in hardware. De logica die nodig is voor communicatie tussen deze modules wordt automatisch door N2C gegenereerd. Dit proces wordt *interfacesynthese* genoemd.

Hardware Alle hardware-modules spreken[1] één (of meerdere) van de CoWare protocollen. Al deze protocollen kunnen m.b.v. een gegenereerde interface vertaald worden naar het busprotocol (bv. Advanced High-speed Bus (AHB)) van het uiteindelijke systeem. Op die manier is het gemakkelijker om een ontwerp aan te passen aan een andere processorkern, ook indien die processor een ander bus gebruikt. Het ontwerpen van de eigen modules is immers onafhankelijk van de gebruikte bus. Via de bus kan de processor (en dus de software) met de hardware-modules communiceren.

Software Bij een softwaremodule met een `master`-poort die met een hardware-module verbonden is, zal de communicatie met de hardware gebeuren via de bus. De processor kan praten met de randapparaten door te lezen van en te schrijven naar geheugenadressen. Men zegt dan dat invoer en uitvoer (I/O) afgebeeld wordt op geheugenadressen. Er worden functies gegenereerd om dit te doen.

Voor een `slave`-poort gebruikt men ook communicatie via geheugenadressen, maar de software wordt nu geactiveerd door een hardwareonderbreking. De softwareroutine zal nu als onderbrekingsroutine (ISR) worden geïmplementeerd.

Als twee softwaremodules met elkaar communiceren, gebruikt men functieoproepen of interprocescommunicatie (IPC) die door een besturingssysteem wordt aangeboden. Als de con-

[1] We gebruiken de uitdrukking: "Twee modules *spreken* een zeker protocol met elkaar," als we bedoelen: "Twee modules gebruiken een zeker protocol om met elkaar te communiceren." Dit naar analogie met mensen die een natuurlijke taal gebruiken om met elkaar te communiceren.

figuratie van de software dat vereist, wordt een ware-tijdbesturingssysteem (*in casu* CoWareOS) ingevoegd.

1.2 Ware-tijdbesturingssystemen

1.2.1 Ware-tijdsystemen, zachte en harde tijdslimieten

We zullen eerst te definiëren wat we bedoelen met een ware-tijdsysteem. Bij ware-tijdsysteem is de tijd waarop uitvoer wordt gegeneerd belangrijk. De invoer komt van de fysische omgeving van het systeem en de uitvoer moet binnen een bepaalde eindige tijdsspanne gegenereerd worden.

Deze tijdsspanne noemen we een tijdslimiet. Men maakt een onderscheid tussen twee soorten tijdslimieten. Bij de eerste soort, de *harde* tijdslimieten, moet de uitvoer zeker binnen de vooropgestelde tijdslimiet gegenereerd worden. Als dit niet lukt, zal het ganse systeem falen. Bij de tweede soort, de *zachte* tijdslimieten, verwacht men wel dat de uitvoer binnen de tijdslimiet wordt gegenereerd, maar het systeem zal nog blijven werken als de tijdslimiet sporadisch niet gehaald wordt.

Er worden speciale besturingssystemen gebruikt om tegemoet te komen aan deze hoge tijdseisen die ware-tijdsystemen opgelegd krijgen. Een besturingssysteem van dit type noemt men een RTOS.

1.2.2 Besturingssystemen voor algemeen gebruik vs. voor ware-tijdsystemen

Er bestaan verschillende besturingssystemen voor algemeen gebruik, zoals MS-Windows en UNIX. Deze besturingssystemen gebruiken vaak technieken die bij een ware-tijdsysteem zinloos of zelfs schadelijk zijn.

Technieken als virtueel geheugen en caches kunnen het systeem wel meer mogelijkheden geven of versnellen, maar maken het zeer moeilijk om uitspraken te doen over de uitvoeringssnelheid van een programma. Ze zullen in ware-tijdsystemen vermeden worden, omdat dit soort

systemen net voorspelbare uitvoeringstijden eist.

Bij een gewoon besturingssysteem kunnen verschillende programma's tegelijk worden uitgevoerd. Zo'n programma dat uitgevoerd wordt, noemt men een proces. De verschillende processen hebben geen volledige controle over het systeem en ze worden tegen elkaar beschermd. Dit is nuttig aangezien de programma's door verschillende programmeurs en met verschillende doelstellingen geprogrammeerd kunnen zijn. De verwachtingen van een gebruiker van de computer kunnen strijdig zijn met sommige doelstellingen van de programmeurs. Daarom hebben processen gescheiden adresruimtes. Als een proces faalt, zal enkel dat ene proces vernietigd worden. Alle andere processen blijven gewoon verder lopen. Twee processen kunnen enkel met elkaar communiceren via de IPC die door het besturingssysteem wordt aangeboden. Op die manier wordt hun communicatie streng gereglementeerd en moeten beide partijen hiervoor toestemming geven.

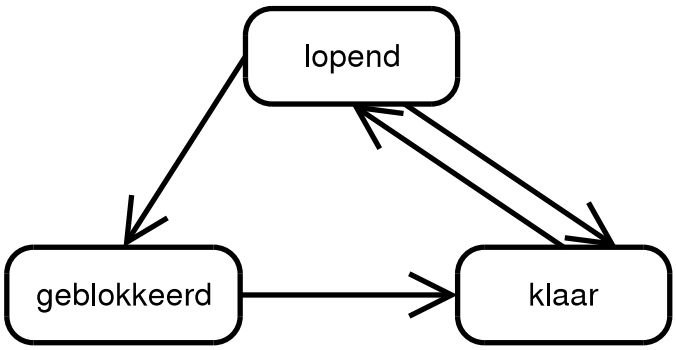
Bij een ingebed systeem is alle software door hetzelfde team of door samenwerkende teams van software-ingenieurs geprogrammeerd. Ze werken niet *tegen* elkaar om hun eigen doelstelling te realiseren, maar *samen* werken ze naar een gemeenschappelijke doelstelling. Bij een update van de software zal *alle* software samen worden aangepast. Er is geen nood om programma's tegen elkaar te beschermen. Indien door een programmeerfout één proces faalt, zal het volledige ingebedde systeem falen. Het heeft dan ook weinig zin om grote inspanningen te doen om de overige processen verder te laten lopen, wanneer één proces faalt.

Omdat processen bij algemene besturingssystemen en bij RTOSen zo veel van elkaar verschillen, noemt men de laatstgenoemde processen *taken*.

1.2.3 Scheduler

De scheduler is het onderdeel van het RTOS dat bijhoudt welke taak op welk moment moet uitgevoerd worden. Het bevat datastructuren die voor elke taak de *toestand* en de *prioriteit* bijhoudt.

De *prioriteit* van een taak wordt meestal aangegeven door een geheel getal. Vreemd genoeg zal in de meeste besturingssystemen een lagere waarde van dat getal een hogere prioriteit aanduiden.



Figuur 1.1: Toestanden van een taak.

Een taak kan zich in één van de volgende *toestanden* bevinden: *klaar*, *geblokkeerd* of *lopend* (zie figuur 1.1)[2]. Als een taak *loopt*, betekent dat dat die taak effectief uitgevoerd wordt door de processor. In een systeem met één processor kan slechts één taak tegelijk in de toestand *lopend* zijn. Een taak maakt de overgangen tussen *lopend* en *klaar* als de scheduler beslist dat dit moet gebeuren. De overgangen worden uitgevoerd door middel van een *taakwissel* (zie 3.2). De lopende taak wordt dan klaar (of geblokkeerd) en één van de taken die klaar is begint te lopen. De overgangen van en naar de toestand *geblokkeerd* noemen we respectievelijk *blokkeren* en *deblokkeren*. Deze overgangen kunnen op een klassiek RTOS veroorzaakt worden door I/O of IPC operaties. We gaan niet in op blokkerende I/O-operaties, maar bespreken enkel IPC

1.2.4 Interprocescommunicatie (IPC)

Taken van een RTOS zouden met elkaar kunnen communiceren via gedeeld geheugen. Dit levert echter een aantal problemen op zoals *wedlopen*. Deze problemen kunnen opgelost worden door gebruik te maken van IPC die het RTOS aanbiedt. Er bestaan verschillende vormen van IPC, zoals semaforen, mutexen, postbussen, wachtlijnen en barrières. Ze hebben met elkaar gemeen dat operaties erop aanleiding kunnen geven tot het blokkeren van een taak. Alle genoemde IPC-primitieven kunnen geïmplementeerd worden met behulp van semaforen. Hiermee wordt bedoeld dat een gebruiker functies kan schrijven die de werking van mutexen, postbussen e.d.

[2] In dit toestandsdiagram en in de tijdlijnen in andere figuren gebruiken we Unified Modeling Language (UML) symbolen.

```
1  void text_producer(void){
2    for(;;){ /*forever*/
3      text=produce_text_to_be_printed();
4      i=next_free_position;
5      copy_text_to_position(text,i);
6      next_free_position = i + 1;
7    }
8  }
```

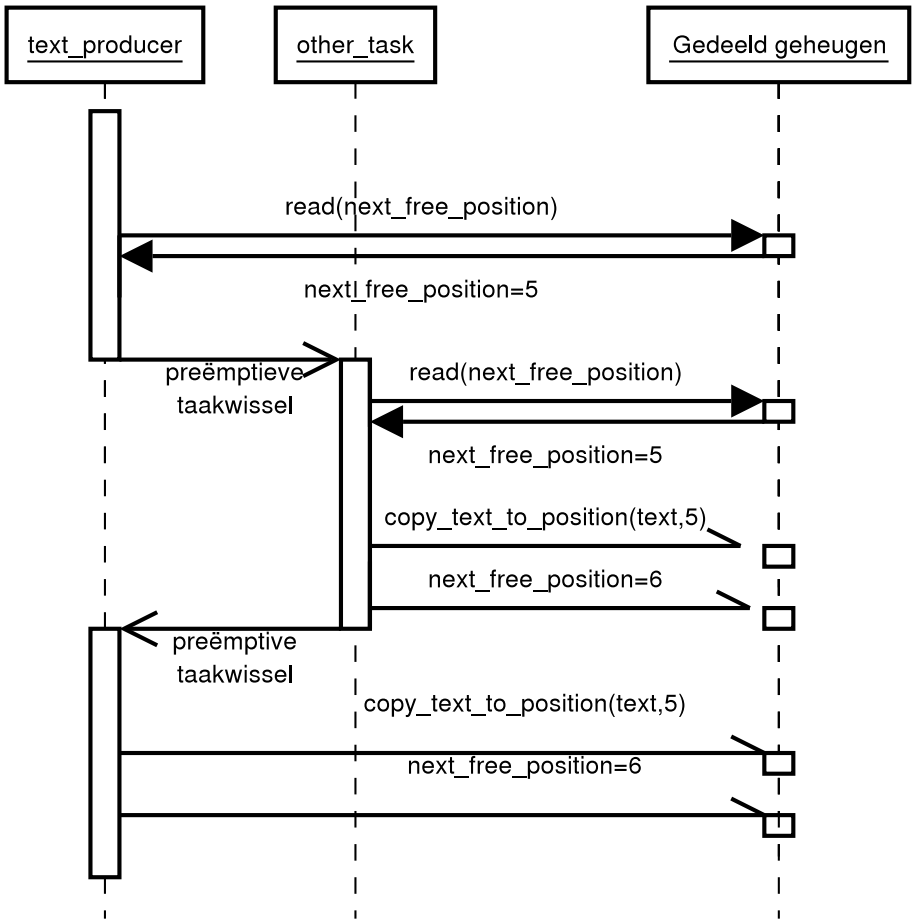
Figuur 1.2: Taak die tekst afdrukt.

simuleren, als hij beschikt over een RTOS dat semaforen ondersteunt. Dit zal evenwel minder efficiënt zijn dan rechtstreekse ondersteuning van deze primitieven door het RTOS. We kiezen ervoor om enkel semaforen te implementeren wegens hun beperkte complexiteit en hun grote bruikbaarheid. Voor meer informatie over de werking en de implementatie van andere IPC-primitieven verwijzen we naar [Lab02, Tan01].

Wedlopen Communicatie via gedeeld geheugen kan problemen opleveren indien geen bijkomende maatregelen genomen worden. Hier volgt een situatieschets van een systeem dat onverwacht gedrag vertoont.

Stel dat een printer in een systeem wordt aangestuurd door één stuurprogramma. Alle andere taken kunnen gegevens in een buffer plaatsen om afgedrukt te worden. Een teller duidt aan waar de eerstvolgende vrije plaats is. Figuur 1.2 bevat een taak die gebruik maakt van de printer. Nu stelt zich het probleem dat we in de tijdlijn van figuur 1.3 geïllustreerd hebben. Het is mogelijk dat de taak onderbroken wordt tussen regel 4 en 5. Als op dat moment een andere taak de printer wil gebruiken, zal ze `next_free_position` lezen, haar data op die positie kopiëren en de verhoogde versie van `next_free_position` terug in het gedeelde geheugen schrijven. Wanneer de eerste taak weer verder mag lopen, zal ze geen rekening houden met wat er gebeurd is, terwijl ze onderbroken was. Ze overschrijft de gegevens van de andere taak en schrijft een foutieve waarde van `next_free_position` in het geheugen. Eigenlijk zou `text_producer` haar gegevens op positie 6 moeten schrijven en de waarde van `next_free_position` op 7 moeten zetten.

We merken op dat compacte C-syntaxis dit probleem niet vermijdt. Afhankelijk van de



Figuur 1.3: Wedren-tijdlijn.

```
void text_producer(void){
  for(;;){ /*forever*/
   text=produce_text_to_be_printed();
    OsSemaWait(printer_sema);
    i=next_free_position;
    copy_text_to_position(text,i);
    next_free_position = i + 1;
    OsSemaSignal(printer_sema);
  }
}
```

Figuur 1.4: Taak die tekst afdrukt met semaforen.

compiler is het goed mogelijk dat de uitdrukking

```
copy_text_to_position(text,next_free_position++);
```

precies dezelfde binaire code oplevert als de code van figuur 1.2.[3] Er wordt onderbroken tussen twee machine-instructies, niet tussen twee lijnen van de hoog-niveauprogrammatekst.

Semaforen Als oplossing voor het probleem van de wedlopen kan men gebruik maken van semaforen. Een semafoor is een gegevensstructuur met een natuurlijke waarde. Er zijn slechts twee bewerkingen mogelijk op semaforen: *verhogen* en *verlagen*. Wanneer een taak een semafoor verlaagt en de waarde van de semafoor was niet nul, zal de taak verder blijven lopen en de waarde van de semafoor wordt met één verminderd. Als de waarde van de semafoor nul was, zal de taak geblokkeerd worden. We zeggen dat de *taak geblokkeerd is op de semafoor*.

Als een semafoor verhoogd wordt en er is een taak geblokkeerd op die semafoor, dan zal die taak gedeblokkeerd worden. Als er geen taak geblokkeerd is op de semafoor, zal bij de waarde van de semafoor één opgeteld worden.

De scheduler zal na een semafoorbewerking eventueel een taakwissel uitvoeren zodat de taak met hoogste prioriteit die *klaar* is, wordt uitgevoerd door de processor.

Semaforen kunnen gebruikt worden om hulpbronnen (gedeeld geheugen, randapparaat,...) te beschermen. Voordat de hulpbron wordt gebruikt, verlaagt de taak eerst een semafoor. Wanneer de taak de hulpbron niet meer nodig heeft, zal ze de semafoor weer verhogen. In de

[3] De ervaren C-programmeur zal inzien dat de ganse romp van de functie `text_producer` zelfs geschreven kan worden als een `for`-instructie met lege romp.

tussentijd zullen alle taken die hetzelfde proberen te doen, blokkeren. Het probleem van figuur 1.2 wordt opgelost met semaforen in figuur 1.4.

1.3 Probleemstelling

In de context van co-ontwerp is het vanzelfsprekend dat de ontwerper voor een module kan beslissen of ze in hardware of in software wordt uitgevoerd. Gewoonlijk zal men het RTOS op een metaniveau beschouwen. Het is als het ware een bijproduct van het genereren van de software. Het RTOS wordt indien nodig ingevoegd door de N2C hulpprogramma's.

Men kan het RTOS echter ook als een gewoon blok beschouwen dat naar keuze in hardware of in software uitgevoerd kan worden. Er zal enkel een kleine hoeveelheid software op het metaniveau blijven. Die software is nodig om de afzonderlijke softwaretaken op te starten en voor de communicatie met de hardware.

In hoofdstuk 2 overwegen we welke implementatievorm we voor het RTOS zullen gebruiken en in welke programmeer- en ontwerptalen dit moet gebeuren. In hoofdstuk 3 bespreken we de door ons geschreven software en de methodes die we gebruiken om taakwissels mogelijk te maken. De daaropvolgende hoofdstukken beschrijven twee implementaties van hardware-modules die de RTOS-functies overnemen en de snelheid die van de tweede implementatie te verwachten valt. We geven in hoofdstuk 6 een opsomming van nuttige mogelijke uitbreidingen aan onze implementatie, waarvan we vermoeden dat ze geen onoverkomelijke problemen opleveren.

Hoofdstuk 2

Een RTOS in hardware

2.1 Hardware vs. software RTOS

Het kan niet *a priori* gegarandeerd worden dat een RTOS in hardware (HWOS) beter zal zijn voor een bepaalde applicatie dan een RTOS in software. De criteria om een implementatie in deze context *beter* te noemen, kunnen ook zeer verschillend zijn. Ze kunnen gebaseerd zijn op uitvoeringssnelheid, energieverbruik, fysische grootte, configureerbaarheid of ontwerpsnelheid.

In deze scriptie onderzoeken we wat de gevolgen zijn van het omzetten van een RTOS in een HWOS.

We verwachten de volgende voordelen van een HWOS:

- Verhoogd parallellisme. De processor zal niet gebruikt worden om te bepalen welke taak moet uitgevoerd worden. Enkel de taakwissels zullen tijd van de processor vragen.
- Snelle afhandeling van onderbrekingen. Sommige onderbrekingen hebben als enige effect dat een taak in de *klaar*-toestand wordt gezet, of dat een teller wordt verhoogd. Het HWOS moet deze onderbrekingen kunnen verwerken zonder dat de processor effectief onderbroken wordt. Enkel indien de zopas gedeblokkeerde taak een hoger prioriteit heeft dan de lopende taak zal het HWOS de processor onderbreken. Het HWOS kan zo ver uitgebreid worden dat het de encoder voor onderbrekingsprioriteiten (IPE) volledig vervangt. (zie paragraaf 4.2)

- Centrale IPC op multiprocessorsystemen. Bij deze systemen vraagt de communicatie tussen verschillende processors via een RTOS extra veel tijd. Dit zou versneld kunnen worden door het gebruik van het HWOS. Eventueel kan het HWOS beslissen of een aanvraag behandeld moet worden door de ene of de andere processor. In deze scriptie is niet ingegaan op multiprocessorsystemen.

- Het HWOS zal aangepast kunnen worden aan het beoogde doel. Er kan geen sprake zijn van een standaard zwarte doos. De systeemontwerper zal de volgende parameters kunnen optimaliseren voor de specifieke eisen van het systeem:

  - Aantal taken en semaforen
  - Welke semaforen voor welke taken gebruikt zullen worden. Met deze informatie kan de taak-semafoormatrix worden geöptimaliseerd (zie paragraaf 4.2.3)

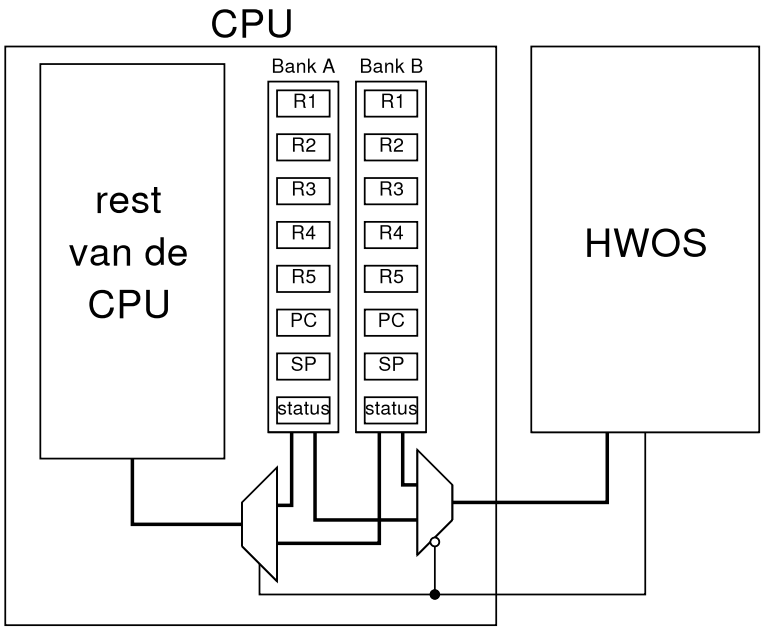
Door middel van simulaties kan men de eigenschappen (snelheid, fysische grootte,...) van het systeem met HWOS en met RTOS gaan meten. Nadien kan men dan kiezen welk van de twee alternatieven best voldoet aan de ontwerpeisen.

Het belangrijkste nadeel van het HWOS is de beperkte flexibiliteit. Naargelang de gekozen architectuur (zie paragraaf 2.2) kan het zeer moeilijk of zelfs onmogelijk blijken om tijdens de werking van het systeem het scheduleralgoritme te veranderen of taken of semaforen aan te maken. Ook updates van de software kunnen problematisch zijn als belangrijke eigenschappen van het HWOS moeten aangepast worden. Maar als het HWOS op een herprogrammeerbare poortmatrix (FPGA) uitgevoerd is, zullen hardwareupdates wel even gemakkelijk zijn als softwareupdates. In vergelijking met een applicatiespecifiek geïntegreerd cicuit (ASIC) zal de chipgrootte, uitvoeringssnelheid en het energieverbruik van een FPGA wel minder goed zijn.

2.2 Architectuur

Een RTOS biedt bepaalde functies aan een ingebed systeem. Er zijn verschillende manieren om deze functies te implementeren:

1. Aangepaste processorarchitectuur



Figuur 2.1: Processor met dubbele registers.

2. Aangepaste microcode

3. Coprocessor

4. Een randapparaat, aangesloten op een bus

5. Een softwareroutine of -bibliotheek

Met uitzondering van de laatste mogelijkheid, zijn dit hardware-implementaties. De laatste mogelijkheid verwerpen we, omdat we in deze scriptie precies proberen af te stappen van software. Bij de overige vormen hebben we uiteraard nog steeds software nodig die die communicatie met het HWOS verzorgt. Het gaat hier echter om een minimale hoeveelheid software.

2.2.1 Aangepaste processorarchitectuur

Men kan een processor ontwerpen die alle registers (ook het statusregister, de stapelwijzer en de programmateller) dubbel heeft (zie figuur 2.1). Via een extra aansluiting zou deze processor een taakwissel kunnen doen in één enkele klokcyclus. Eventueel zal er een klein tijdsverlies zijn als de pijplijnen en caches moeten gewist worden. Tijdens de werking van de processor kunnen de

niet-actieve registers gelezen en overschreven worden. Indien deze bewerkingen gebeuren via een extra aansluiting aan de processor, zal zelfs de bus niet bezet worden.

Dit soort processor is zeer nuttig als een HWOS (of een andere processor) de processor kan controleren. Bij een processorchip probeert men het aantal pinnen te minimaliseren. Het aanbrengen van extra aansluitingen is hier zeker niet gewenst. Bij een processorkern die samen met de rest van het systeem op één chip is geïntegreerd stelt dit probleem zich veel minder.

Er zijn enkele nadelen verbonden aan dit voorstel. De investering voor het ontwerpen van deze processorkern kan hoog oplopen. Een processor is immers een veel complexer object dan het eenvoudige HWOS waarover deze scriptie handelt. Ook zal het ontdubbelen van alle registers de grootte van de processorkern nadelig beïnvloeden. Men moet ook opletten dat de snelheid van de registers niet te sterk degenereert.

In het kader van een scriptie is het niet haalbaar een ganse processor te ontwerpen.

2.2.2 Aangepaste microcode

Sommige processors (vooral CISCs) gebruiken intern *microcode* om hun instructieset te implementeren. Om één machine-instructie uit het geheugen te laden en uit te voeren zullen binnen in de processor meerdere micro-instructies worden uitgevoerd.

AJILE SYSTEMS (webstek: [AJI]) biedt op het moment van dit schrijven al processorchips en IP-kernen aan die een gemicroprogrammeerd RTOS hebben.

De voordelen van deze implementatievorm zijn:

- Snelheid: er moet maar één enkele instructie uit het geheugen opgehaald worden om een RTOS-aanroep te doen.
- Indien uitgegaan wordt van een bestaande gemicroprogrammeerde processor, is de implementatie relatief eenvoudig. Er moet immers enkel aangepaste microcode geschreven worden, op voorwaarde natuurlijk dat er genoeg microcode-ROM is voor de toegevoede code.
- Men kan in een C-programma een RTOS-oproep doen door een macro te gebruiken die enkele machine-instructies tussenvoegt.

- Het RTOS is wel schaalbaar, in tegenstelling tot de implementaties die een apart geheugen voorzien voor de interne gegevensstructuren. De implementatie als randapparaat op de bus is hier een voorbeeld van.

Het nadeel echter is dat er is geen parallelisme toegevoegd wordt. De processor zal voor complexere bewerkingen nog steeds meerdere klokcycli nodig hebben. Indien, zoals waarschijnlijk is, de nodige datastructuren in gewoon RAM-geheugen bewaard worden, zijn ook meerdere geheugentoegangen nodig. In zekere zin is dit gewoon een stap in de richting van CISC, waar bepaalde veelvoorkomende routines gemicroprogrammeerd zijn om de snelheid te verhogen. Men verlegt in feite de software van machinecode naar microcode.

Er is bij deze thesis vooropgesteld dat we zouden werken met een bestaande processorkern. Hoewel het zeer interessant zou zijn een onderzoek te voeren in de richting die gekozen is door AJILE SYSTEMS, zullen we deze implementatievorm verwerpen.

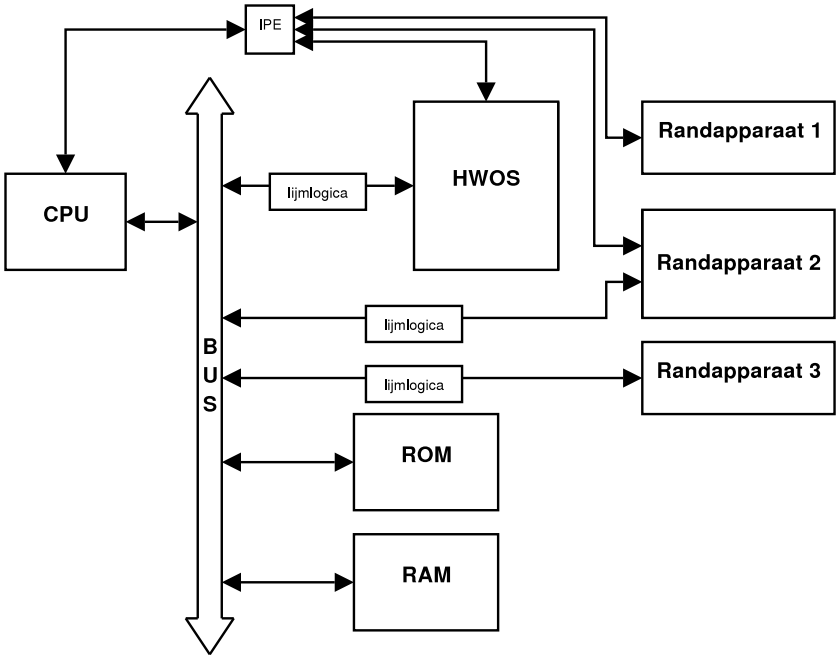
2.2.3 Coprocessor

Men kan de processor ongewijzigd laten en het HWOS implementeren in een coprocessor. Men behoudt het voordeel dat een RTOS-oproep slechts één of enkele machine-instructies kost. Men kan echter een bestaande processorkern behouden en de coprocessor kan worden hergebruikt voor verschillende processors van dezelfde familie.

We hebben onderzocht of deze implementatievorm haalbaar is voor de thesis. Er stellen zich echter een aantal problemen:

- Er is bijs CoWare-Leuven geen ervaring met deze manier van ontwerpen.
- De N2C hulpprogramma's zijn niet ontwikkeld om te co-simuleren met coprocessors. Het zou vrij veel werk vragen om N2C aan te passen voor dit doel.
- Een coprocessor is enkel overdraagbaar naar andere processors van dezelfde familie. Dit is een sterke beperking.

Indien het ontwerpen van een coprocessor zou worden ondersteund in N2C zou dit een goed alternatief zijn. Onder de gegeven omstandigheden is de investering om een coprocessor te



Figuur 2.2: Het HWOS als randapparaat op de bus.

implementeren en in de N2C ontwerpstroom in te passen, te groot. We zien ook af van deze implementatievorm.

2.2.4 Randapparaat op de bus

We kiezen ervoor het HWOS te implementeren als randapparaat, zoals geïllustreerd in figuur 2.2. Dankzij de interfacesynthese van CoWare kan deze implementatie onafhankelijk gemaakt worden van de gebruikte bus. Indien men het HWOS overbrengt naar een andere processor en een andere bus, zal enkel het softwaregedeelte moeten worden herschreven. Dit gedeelte zal, zoals aangegeven in hoofdstuk 3, minimaal zijn. Het herschrijven zal dus geen grote inspanning meer vergen.

Het is tevens de gemakkelijkste manier om een eerste werkend prototype van een HWOS te maken. Later kan het HWOS eventueel geherimplementeerd worden als coprocessor of als microcoderoutines. Er kan later ook een processor ontworpen worden zoals degene die in paragraaf 2.2.1 beschreven wordt. Het HWOS zal vrij gemakkelijk aangepast kunnen worden om met zo een processor te werken.

2.3 Ontwerpprogramma's en talen

2.3.1 Hardware: CoWareC / N2C

Om de hardware te ontwerpen maken we gebruik van N2C van CoWare. In hoofdstuk 1.1 hebben we de ontwerpmethodologie van CoWare besproken. N2C bestaat uit een verzameling ontwerpprogramma's die de ontwerptalen VHDL, Verilog en CoWareC ondersteunen. De belangrijkste eigenschappen van N2C voor ons ontwerp zijn:

- Co-simulatie van hardware en software
- Mogelijkheid tot het symbolisch ontluizen van de hardware, gebaseerd op Gnu DeBugger (GDB) en Data Display Debugger (DDD), cf. webstek [gdb, ddd]. Software kan jammer genoeg enkel ontluisd worden met de ontluizer van ARM Developer Suite (ADS). Deze is (veel) minder krachtig dan de GDB.
- De bus en de lijmlogica worden automatisch gegenereerd door de N2C interfacesynthese.

N2C heeft ook enkele nadelen:

- De hardware-ontwerptaal SystemC ([sys02]) wordt niet (bruikbaar) ondersteund. We hebben alle hardware geprogrammeerd in CoWareC. Deze taal is eigendom van CoWare en vertoont sterke overeenkomsten met SystemC. CoWare is van plan om op korte termijn van deze taal af te stappen en over te stappen naar SystemC.
- Sommige onderdelen van het N2C-pakket zijn niet zo stabiel als men zou mogen verwachten.

2.3.2 Software: ISO C en assembler code

Het is mogelijk om de software te laten generen vanuit de CoWareC brontekst. N2C zal echter code toevoegen en eventueel een RTOS invoegen. In ons geval is dit echter niet nuttig omdat we zelf alle controle willen behouden over de hulpfuncties en over het RTOS. We implementeren immers een vervanging voor het bestaande RTOS. De software voor de processorkern is geschreven in C en in assemblercode.

We hebben ons zoveel mogelijk aan de ISO C standaard gehouden, maar soms hebben we uitbreidingen moeten gebruiken die tegen de standaard ingaan.

We gebruiken het taalwoord `inline` bij sommige functiedefinities om sneller de functies uit te voeren. Het effect hiervan is dat de programmacode van de functie wordt tussengevoegd op alle plaatsen waar die functie wordt opgeroepen. Op die manier wordt de vertraging ten gevolge van de functieoproep geminimaliseerd. `inline` is een woord uit C++ (zie [Str97]) en wordt niet standaard ondersteund in C. Indien dit taalwoord niet ondersteund wordt door de gebruikte C-compiler, kan men het weglaten zodat de functie-oproepen zoals gewoonlijk met een sprong worden geïmplementeerd, ofwel kan de functie als preprocessor macro worden geïmplementeerd.

We gebruiken de `asm` instructie om assemblercode in een C-programmatekst te kunnen insluiten. Hoewel `asm` standaard ondersteund is in C++, maar niet in C, vermoeden we dat de meeste C compilers een soortgelijke instructie ondersteunen.

Enkele routines zijn in assemblercode geprogrammeerd. Het gaat om een herimplementatie van `setjmp` en `longjmp` en een onderbrekingsroutine voor taakwissels. We hebben enkel een versie geprogrammeerd voor Advanced RISC Machines (ARM). Indien men het HWOS met een andere processor wil gebruiken, zullen deze routines moeten worden aangepast.

Hoofdstuk 3

Software

3.1 Communicatie met hardware

Zoals uitgelegd in paragraaf 1.1.2, wordt de communicatie tussen software- en hardware-modules afgebeeld op geheugenadressen. Wanneer we zelf de software willen schrijven en deze dus niet laten genereren door N2C, zullen we ook routines moeten voorzien voor deze communicatie. Aangezien de communicatie afgebeeld is op geheugenadressen, zal hardwarecommunicatie eenvoudig kunnen gebeuren door te lezen van of te schrijven naar die adressen. In C-jargon noemt men dit het het dereferentiëren van *pointers*.

De hardware weet wanneer er gelezen en geschreven wordt naar deze adressen en zal nuttige neveneffecten kunnen toekennen aan deze bewerkingen. Zo zal een geheugentoegang naar de `semaSignal`-poort als effect hebben dat een semafoor intern in het HWOS verhoogd wordt.

3.2 Taakwissels: `setjmp`, `longjmp`

Om een taakwissel uit te voeren moeten de registerinhouden van de processor opgeslagen worden in het geheugen en worden de nieuwe registerwaarden uit het geheugen geladen. Op die manier kan de processor een andere taak beginnen uitvoeren en kan later de uitvoering van de eerste taak worden hervat. Om taakwissels te implementeren gebruiken we `setjmp` en `longjmp`.

```
/* ----------------setjmp_demo.c -------------*/
#include <setjmp.h>
#include <stdio.h>
int main(){
  jmp_buf env;
  int i;
  if(!(i=setjmp(env))){ /*assignment!*/
    printf("after setjmp \n");
    printf("returned value %d \n",i); /*i==0*/
    longjmp(env,20);
  }else{
    printf("after longjmp \n");
    printf("returned value %d \n",i); /*i==20*/
  }
  printf("finished\n");
}
```

Figuur 3.1: `setjmp` voorbeeldprogramma.

`setjmp` en `longjmp` In de ISO C standaard worden `setjmp` en `longjmp` gedefinieerd als macro's. In sommige bibliotheken (zoals de GNU C-standaardbibliotheek) worden ze echter eerst als functies geïmplementeerd en daarna worden deze functies geherdefinieerd als macro's. We zullen ze dan ook als functies beschouwen, waarvan de definities gegeven worden door:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

`setjmp(env)` heeft als effect dat de registers worden opgeslagen in de gegevensstructuur `env` (we noemen dit de *omgeving*). Zoals blijkt uit de functiedefinitie, is `env` van het gegevenstype `jmp_buf`. De teruggegeven waarde van `setjmp` is altijd nul.

`jmp_buf` kan ofwel geïmplementeerd worden als gegevensstructuur (dit is b.v. zo op ARM) of als rij (dit is zo op HP-UX). Dit geeft een probleem om op een overdraagbare manier het adres van de omgeving te bepalen. In het eerste geval is `(int)&env` een adres, en in het tweede geval is `(int)env` een adres. We lossen dit op door een `jmp_buf` als eerste element te nemen van een gegevensstructuur, die we `jmp_buf_s` noemen. Als `env_s` van het type `jmp_buf_s` is, zal het adres in elke implementatie `(int)&env_s` zijn. We willen dat de C-brontekst zo gemakkelijk mogelijk overdraagbaar is naar andere platforms. Dit maakt het ook mogelijk de software te

ontluizen op de ontwikkelingscomputer (b.v. op i686-PC), en niet enkel op een gesimuleerde ARM processor. Een gesimuleerde processor zal immers veel trager zijn dan de processor van de ontwikkelingscomputer.

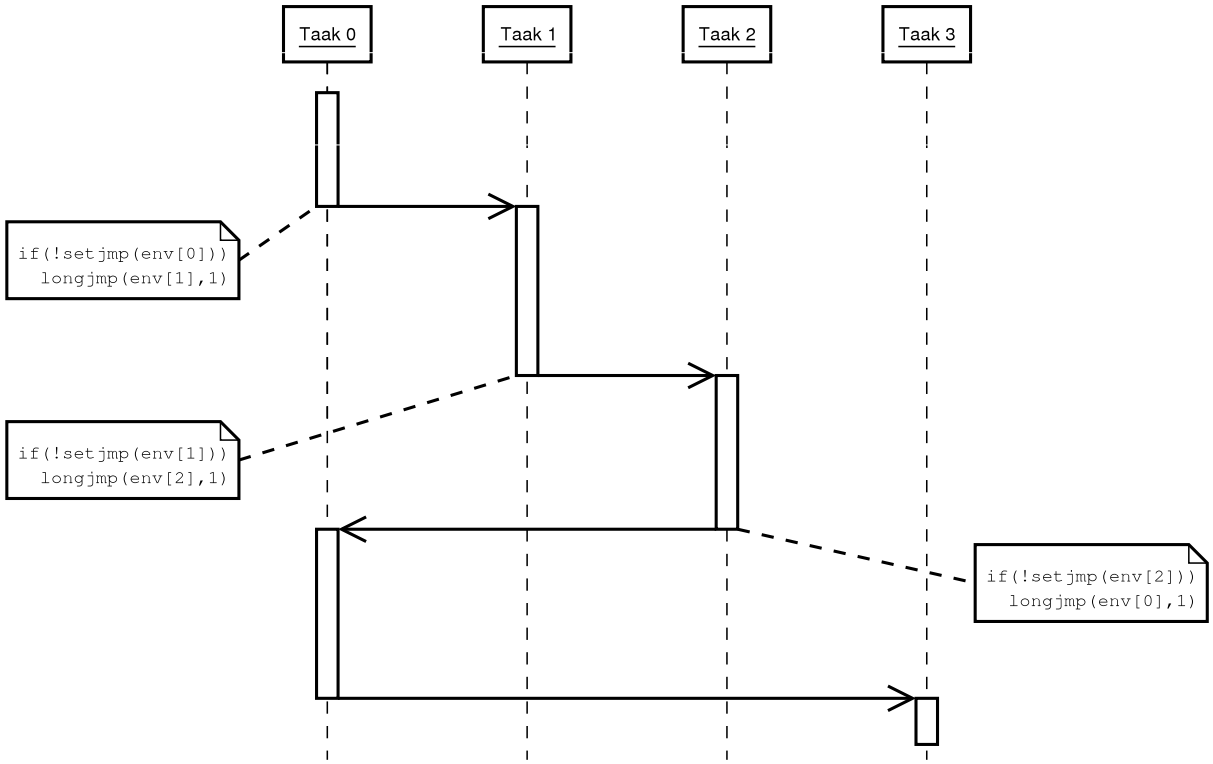
De registers, opgeslagen door `setjmp`, kunnen op een later tijdsstip hersteld worden door de opdracht `longjmp(env,val)` uit te voeren. Het effect hiervan is dat de registerwaarden uit de omgeving weer in de registers worden geladen. Het register dat de teruggegeven waarde van de functie `setjmp(env)` moet bevatten (b.v. `r0` op ARM processors) wordt niet hersteld, maar krijgt per definitie de waarde `val?val:1`. In mensentaal betekent dit: "Als de waarde van `val` niet nul is, wordt die waarde teruggegeven, anders wordt één teruggegeven." Het is dus onmogelijk dat deze waarde nul is. Na het oproepen van `longjmp` zal het programma voortgaan op de plaats waar `setjmp` voor het laatst werd opgeroepen. Aan de hand van deze registerwaarde kan het programma weten of er net een *echte* oproep van `setjmp` is uitgevoerd (waarde is nul), dan wel teruggesprongen is door `longjmp` (waarde verschilt van nul). Figuur 3.1 bevat een kort voorbeeld van het gebruik van `setjmp` en `longjmp`.

Het moet opgemerkt worden dat sommige implementaties *niet alle* registers opslaan. Bij conventie worden op de ARM architectuur de registers `r0-r3` gebruikt voor parameterdoorgave bij functie-oproepen. De waarde van het statusregister heeft geen betekenis na een functieoproep. Deze vijf registers worden door de functie `setjmp` dus niet opgeslagen. In de `jmp_buf` gegevensstructuur is er ook geen plaats voorzien voor deze registers. Meer informatie over deze functies kan men vinden in elk goed boek over C, b.v. [KP98].

Taakwissels Veronderstel nu dat we een rij `env[]` van vier omgevingen hebben. Elk van deze omgevingen hoort bij een taak die uitgevoerd wordt. Een taak kan nu de controle doorgeven aan een andere taak door haar eigen registers in het hoofdgeheugen op te slaan en de registerwaarden van de andere taak uit het geheugen op te halen. In C kan dit verwezelijkt worden door:

```
if(!setjmp(my_env)){
  longjmp(other_env,1);}
```

Op die manier hebben we een mogelijkheid om taakwissels uit te voeren. Figuur 3.2 geeft aan hoe vier taken de controle aan elkaar kunnen doorgeven.



Figuur 3.2: Taakwissels met `setjmp` en `longjmp`

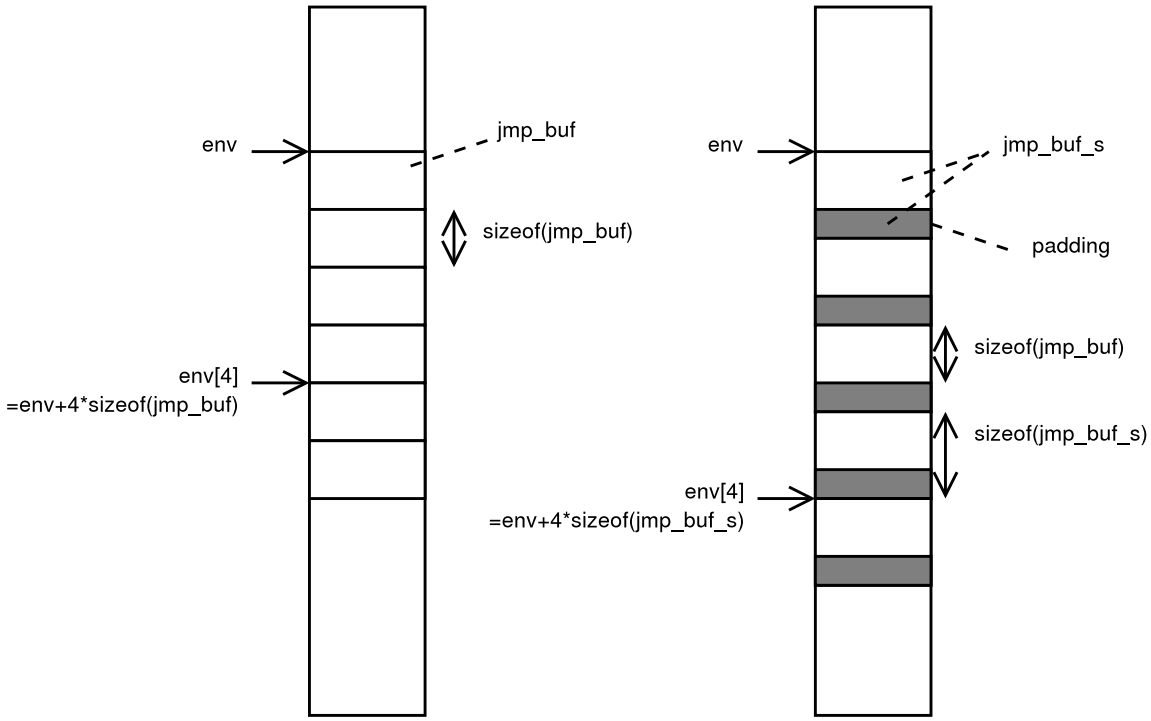
Als een hardware-module als scheduler gebruikt wordt, kan een taak uit zichzelf niet weten aan welke taak de controle moet worden overgedragen. Er moet aan de hardware gevraagd worden op welke adressen `my_env` en `other_env` zich bevinden.

Het creëren van de verschillende omgevingen is meer dan enkel kopieën maken van één omgeving. De verschillende taken mogen niet op elkaars stapel schrijven. Men moet in de omgeving de positie kennen van de stapelwijzer en de inhoud van de stapelwijzer doen wijzen naar een locatie die voorzien is voor de stapel van die taak. We merken op dat de positie van de stapelwijzer implementatieafhankelijk is. Deze positie zal –per platform– opgenomen moeten worden in een hoofdingbestand. Ze kan best bepaald worden door de assembler-broncode van de specifieke implementatie te bestuderen.

Taken opstarten De routine om de taken op te starten is niet al te complex, maar wel moeilijk te programmeren en te ontluizen. Dit komt doordat deze routines geprogrammeerd wordt op een lager niveau dan het niveau van de programmeertaal C. Er worden immers bewerkingen (bijna) rechtstreeks op de registers gedaan. Een ontluizer biedt hier slechts in zeer beperkte mate hulp.

Ons eerste idee was om een functie `OsFork` te maken die de gegevensstructuren voor een nieuwe taak opzet, maar die onmiddellijk terugkeert naar de oproeper, zonder eerst een deel van de nieuwe taak uit te voeren. Deze implementatie faalde. Het probleem was dat we een omgeving creëerden in de `OsFork` functie en probeerden na het beëindigen van die functie terug te springen naar die omgeving. Dit is expliciet niet toegelaten door `setjmp` en `longjmp`. Door de structuur van SPARC processors *kan* deze techniek toch lukken, maar vroeg of laat zal ze op onvoorspelbare wijze falen. De simulatie op een SPARC-processor was niet overdraagbaar naar ARM. Hoewel het, mits wat laag-niveauprogrammeerwerk, wel mogelijk moet zijn om de beschreven functie stabiel te maken op een willekeurig platform, hebben we een andere methode gekozen om alle taken op te starten.

De opstartroutine kent voor elke taak de functie die moet gebruikt worden om de taak op te starten. Onze definieve implementatie zal voor elke taak een omgeving aanmaken met een eigen stapel. Alle omgevingen hebben een programmawijzer met dezelfde waarde. Dat betekent dat, wanneer een taak door longjmp geäctiveerd wordt, de taak zelf niet weet welke functie ze



Figuur 3.3: Spatiëring van omgevingen.

moet uitvoeren. De taak vraagt dan aan het HWOS welk taaknummer ze heeft en zal dan de startfunctie van met dat taaknummer uitvoeren.

De brontekst van de procedure die we gebruiken om alle taken op te starten kan gelezen worden in het bestand `hwos.c`.

3.3 Spatiëring van omgevingen

Indien het HWOS het nummer van een taak (`tasknumber`) zou doorgeven aan de software, kan de software gemakkelijk het adres van de omgeving van die taak vinden door de volgende berekening uit te voeren (zie ook figuur 3.3):

```
env+sizeof(jmp_buf)*tasknumber
```

De programmeur zal enkel een rij indexeren (`env[tasknumber]`) en de compiler genereert code die de bovenstaande berekening uitvoert. Om de uitvoeringstijd op de processor te minimaliseren verkiezen we deze berekening in hardware te laten uitvoeren.

Indien het binaire beginadres van de omgevingen zo wordt gekozen dat het eindigt op voldoende nullen, kan men deze berekening vereenvoudigen tot een aaneenschakeling van bitrijen:

$$\langle basis \rangle . \langle taaknr \rangle . \langle suffix \rangle$$

De lengte van $\langle suffix \rangle$ is $\lceil log_2(\text{sizeof(jmp\_buf)}) \rceil$ waarbij `sizeof(jmp_buf)` de grootte van de datastructuur, die de omgeving bevat, is.[1] Indien deze grootte geen gehele macht van twee is, wordt de omgeving uitgebreid totdat de totale grootte wel een gehele macht van twee is. De lengte van $\langle taaknr \rangle$ is $\lceil log_2(aantal\_taken) \rceil$. Het beginadres (`env`) van de omgevingen is $\langle basis \rangle$, aangevuld met nullen. Bijvoorbeeld, als `env=0x1ff70000` dan is $\langle basis \rangle$`=0x1ff7`, en is `env[`**`5`**`]=0x1ff70`**`5`**`00`.

We laten dus een deel van het hoofdgeheugen ongebruikt om rekentijd uit te sparen. In de brontekst van het HWOS gebruiken we de `jmp_buf_s` gegevensstructuur i.p.v. `jmp_buf`. `jmp_buf_s` bevat `jmp_buf` plus een opvulsel zodat de totale grootte van `jmp_buf_s` een gehele macht van twee is.

3.4 Preëmptie

Preëmptie heeft te maken met het moment waarop de uitvoering van een taak (tijdelijk) wordt stopgezet[2]. Dit komt overeen met een toestandsverandering van de taak van toestand *lopend* naar *geblokkeerd* of *klaar*.

Sommige besturingssystemen zullen de uitvoering van een taak enkel stoppen als de taak zelf een OS-functie oproept. Dit kan bijvoorbeeld een semafoorbewerking zijn. In dit geval zegt men dat er *geen preëmptie* is. Merk op dat een taak die in een oneindige lus raakt, nooit door het besturingssysteem zal gestopt worden. Het kan nuttig zijn om een taak te stoppen op andere momenten, bv. wanneer de taak al een bepaalde tijd loopt of wanneer er een signaal van een randapparaat ontvangen wordt. Dit noemt men preëmptie. Preëmptie wordt geïmplementeerd door een onderbreking naar de processor te sturen. De onderbrekingsroutine zal dan

[1] $\lceil x \rceil$ is het kleinste gehele getal dat niet kleiner is dan x.

[2] We gebruiken de term *onderbreken* hier niet omdat we die voorbehouden voor wat in het Engels *interrupt* wordt genoemd.

een taakwissel uitvoeren.

Zoals uitgelegd in paragraaf 3.2 is het mogelijk dat bij een systeem zonder preëmptie[3] niet alle registers worden opgeslagen in de omgeving. Wanneer we preëmptie willen ondersteunen, zullen ook deze registers moeten worden opgeslagen en indien nodig hersteld worden. Aangezien noch de gebruikte implementatie van `setjmp` en `longjmp` noch de `jmp_buf` gegevensstructuur voorzien zijn om nog extra registers op te slaan, hebben we een volledig nieuwe implementatie van `setjmp` en `longjmp` geschreven in assemblercode. Er moet op gewezen worden dat onze implementatie niet voldoet aan de ISO C standaard. Zo zal de teruggegeven waarde na het uitvoeren van `longjmp` steeds één zijn, onafhankelijk van de waarde van de tweede parameter van `longjmp`. We wijken bewust af van de standaard om de uitvoeringssnelheid te verhogen.

Er is ook een onderbrekingsroutine in assemblercode geschreven om taakwissels uit te voeren. Deze routine maakt gebruik van dezelfde datastructuur die de aangepaste `setjmp` en `longjmp` gebruiken. De drie routines zijn dus volledig combineerbaar. Hun assemblercode is opgenomen in bijlage B.

[3] Dit is o.a. het geval bij CoWareOS.

Hoofdstuk 4

Implementaties van het HWOS

We hebben twee verschillende implementaties van een HWOS gemaakt. De functionaliteit van de beide is licht verschillend. De eerste implementatie (HWOS-1) is verder verfijnd dan de tweede implementatie (HWOS-2) en is op details[1] na synthetiseerbaar tot hardware. Het HWOS-2 bevat meer functies dan het HWOS-1 en heeft een volledig andere architectuur.

4.1 HWOS-1

4.1.1 Gedrag

Het HWOS-1 biedt de volgende functionaliteit aan.

Prioriteitsscheduler De scheduler kiest steeds een taak met zo hoog mogelijke prioriteit om uit te voeren. Een taak zal nooit uitgevoerd worden wanneer er een taak met een hogere prioriteit *klaar* is.

Semaforen Taken kunnen met elkaar communiceren via semaforen. Semafooroperaties kunnen taken blokkeren.

Aanpasbare prioriteiten De prioriteiten kunnen dynamisch worden aangepast door de software. Een aanpassing van een prioriteit zal geen zeer snelle operatie zijn, aangezien de interne gegevensstructuren van het HWOS-1 opnieuw moeten gesorteerd worden.

[1] bv. geheugenblokken moeten vervangen worden door geheugenblokken uit een bibliotheek.

We leggen de volgende beperkingen op om de complexiteit van het HWOS-1 laag te houden.

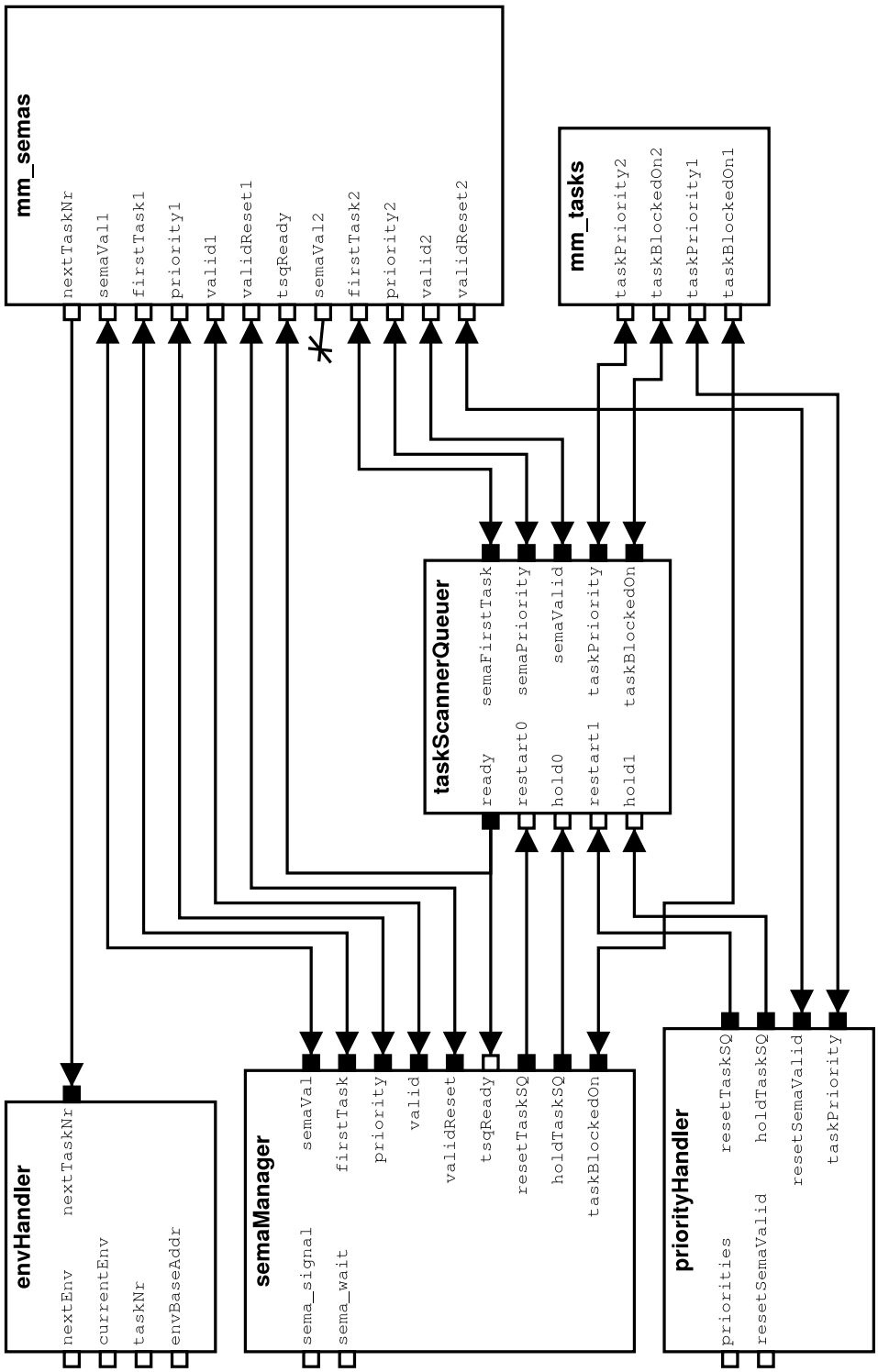
- Het aantal taken dat ondersteund wordt, is beperkt en wordt vastgelegd bij de generatie van het HWOS-1.
- Als twee taken dezelfde numerieke prioriteit hebben, zal het HWOS-1 een arbitraire, maar deterministische keuze maken tussen deze twee taken. Er zal dus niet afgewisseld worden tussen deze twee taken: één van de beide taken krijgt steeds prioriteit boven de andere taak.
- Taken worden enkel onderbroken op eigen aanvraag. Er is dus geen preëmptie.

4.1.2 Structuur en werking

Figuur 4.1 geeft de interne structuur weer van het HWOS-1. De poorten die links niet aangesloten zijn, zijn uitwendige poorten van de module. De poort die met een kruis verbonden is (`mm_semas/semaVal2`) wordt niet gebruikt. De zwart gekleurde poorten zijn `master`-poorten, de witte zijn `slave`-poorten. Connectors met één pijl verbinden `in` en `out`-poorten. De pijlen wijzen van de `out`-poorten naar de `in`-poorten. Connectors met twee pijlen verbinden twee `inout`-poorten.

Men kan de modules indelen in drie klassen:

1. De eerste klasse bevat de modules `envHandler`, `semaManager` en `priorityHandler`. Ze communiceren met de software over de bus en bevatten lijmlogica.
   De `priorityHandler` verwerkt aanvragen van de software om de prioriteit van een taak aan te passen.
2. De modules `mm_semas` en `mm_tasks` bevatten geheugens en schakelelementen. `mm_semas` bevat van elke semafoor de waarde en de taak met de hoogste prioriteit die erop geblokkeerd is. `mm_tasks` bevat van elke taak de prioriteit en het nummer van het semafoor waarop de taak geblokkeerd is (nul indien de taak niet geblokkeerd is). Elke poort van een intern geheugen (stel: `fooBar`) wordt door een schakelelement opgesplitst in twee poorten (stel: `fooBar1` en `fooBar2`). De eerste van deze poorten krijgt steeds voorrang.



Figuur 4.1: Structuur van het HWOS-1.

Als een geheugentoegang via `fooBar1` toekomt, selecteert het schakelelement deze poort, anders wordt de poort `fooBar2` geselecteerd.

3. De laatste klasse bestaat uit slechts één module, `taskScannerQueuer`. Deze module bevat een pijplijn van drie niveaus. De functie van de module is om het geheugen te overlopen en voor elke semafoor de taak met de hoogste prioriteit te zoeken die op die semafoor geblokkeerd is. Er wordt ook gezocht naar de taak met de hoogste prioriteit die klaar is om uitgevoerd te worden. Deze zoekbewerking neemt een aantal klokcycli in beslag, nl. evenveel als er taken zijn plus drie. De extra drie klokcycli worden veroorzaakt doordat deze module een pijplijn van drie niveaus bevat. Indien de inhoud van de geheugens wijzigt tijdens de zoekoperatie, zal deze opnieuw moeten beginnen. Hiervoor dienen de `restart0` en `restart1`. Wanneer `taskScannerQueuer` klaar is, zal een signaal op poort `ready` gegeven worden.

Een meer uitgebreide bespreking van het HWOS-1 wordt opgenomen in een apart verslag dat voor het opleidingsonderdeel DIGITALE COMPONENTEN EN SCHAKELINGEN 2 gemaakt is. Strikt genomen is dit geen onderdeel van deze scriptie en zal het HWOS-1 verder niet besproken worden.

4.2 HWOS-2

4.2.1 Gedrag

In de tweede implementatie van het HWOS (het HWOS-2) hebben we meer parallellisme gebracht. Het geheugen en de logica zijn meer verweven en worden niet in afzonderlijke blokken ondergebracht, zoals in het HWOS-1.

We hebben de volgende functionaliteit:

- Prioriteitsscheduler
- Semaforen
- Preëmptie

- Opvangen van hardwaresignalen (paragraaf 4.2.2)

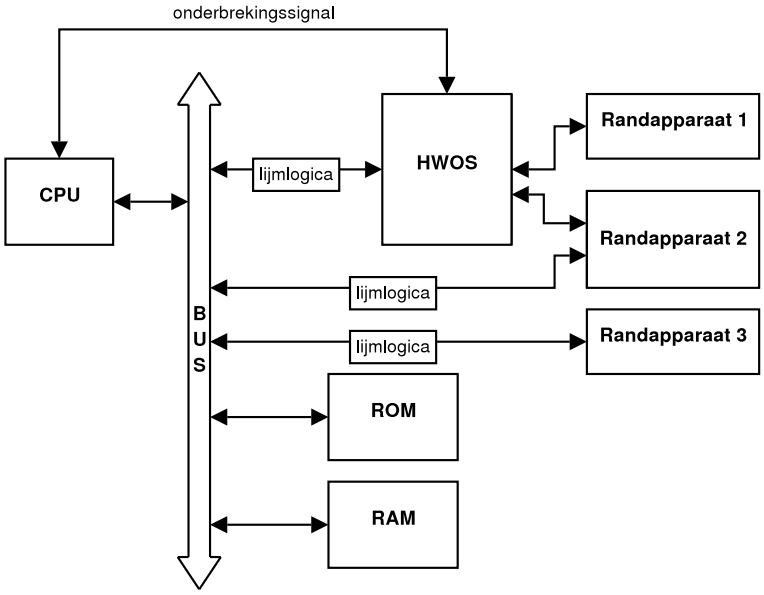
In vergelijking met het HWOS-1 leggen we nog meer beperkingen op:

- De prioriteiten kunnen niet aangepast worden
- De waarde van de semaforen kan niet door de software gelezen worden. Het HWOS-2 is uitbreidbaar om dit wel toe te laten, maar de complexiteit van de hardware zal hierdoor toenemen. Enkel de bewerkingen verhogen en verlagen zijn toegelaten op semaforen.
- Als een semafoor *overloopt*, wordt dit niet door het HWOS-2 opgevangen. Dit gebeurt ook niet bij het HWOS-1, maar daar kent de software de waarde van de semafoor. In het HWOS-2 is die waarde niet gekend en kan het gebrek aan afhandeling door de hardware problemen opleveren. Men kan wel de maximale waarden van de semaforen kiezen bij het genereren van het HWOS. Dit is echter nog steeds geen garantie dat een semafoor niet zou kunnen overlopen. In paragraaf 6.2.1 wordt een mogelijke afhandeling van semafooroverlopen besproken.

4.2.2 Verwerking van hardwaresignalen

Randapparaten kunnen onderbrekingen naar de processor sturen. Als een onderbreking ontvangen wordt, zal een onderbrekingsroutine uitgevoerd worden. Klassiek zal deze routine zo kort mogelijk gemaakt worden om de reactietijd van het systeem niet nadelig te beïnvloeden. Tijdens het uitvoeren van de onderbrekingsroutine zullen andere onderbrekingen immers moeten wachten. Sommige onderbrekingsroutines hebben als enige effect dat de waarde van een gehele variabele wordt verhoogd, of dat een taak wordt gedeblokkeerd.

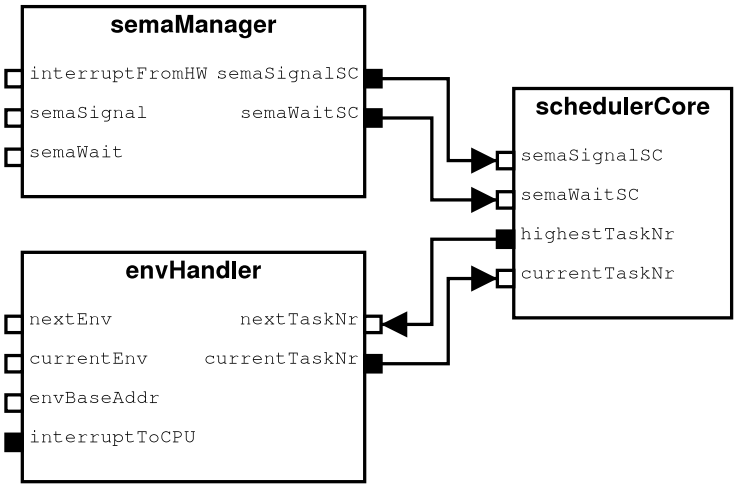
We zullen de onderbrekingssignalen van de randapparaten niet door de IPE maar door het HWOS laten verwerken. Vergelijk hiervoor figuur 2.2 met 4.2. Het HWOS-2 zal zo'n signaal vertalen in het verhogen van een semafoor. Stuurprogramma's[2] zijn dan eenvoudig te schrijven met behulp van semaforen. Een voorbeeld hiervan is opgenomen in figuur 4.3.



Figuur 4.2: HWOS-2 op de bus vangt hardwareonderbrekingen op.

```
void driver(void){
  for(;;){
    OsSemaWait(peripheralSema); /*wacht op signaal van randapparaat*/
    doSomething();              /*communiceer met randapparaat*/
  }
}
```

Figuur 4.3: Stuurprogramma voor gebruik met HWOS-2.



Figuur 4.4: Structuur van HWOS-2.

4.2.3 Structuur en deelblokken

Figuur 4.4 toont de interne structuur van het HWOS-2. Zoals bij het HWOS-1 zijn de niet aangesloten poorten externe poorten. `interruptFromHW` zal verbonden worden met randapparaten en `interruptToCPU` zal onderbrekingen van de software initiëren. De overige externe poorten worden op geheugenadressen afgebeeld en communiceren met de software.

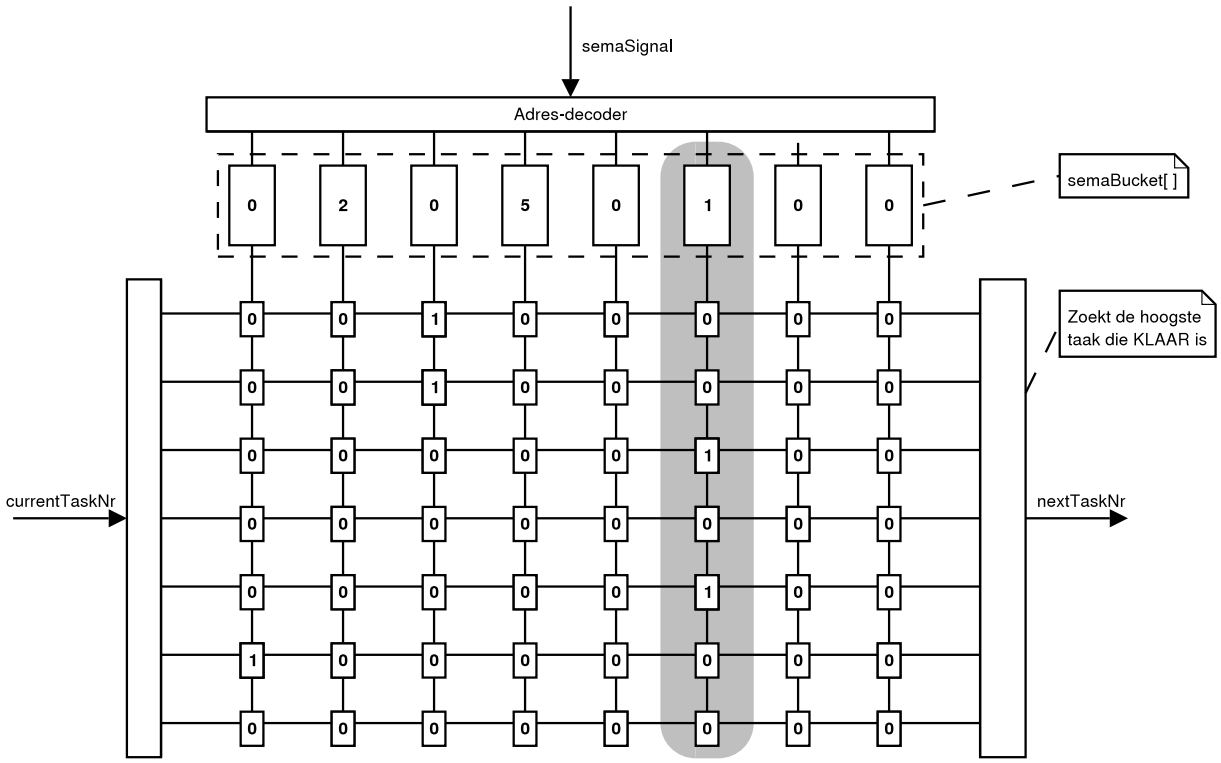
De modules `semaManager` en `envHandler` bevatten lijmlogica en `schedulerCore` berekent welke taak uitgevoerd moet worden.

`semaManager` Deze module vertaalt `semaSignal` en `interruptFromHW` naar de `semaSignalSC`-poort. Dit signaal veroorzaakt een *verhoog*-operatie op een semafoor. De signalen van de `semaWait`-poort worden doorgegeven naar `semaWaitSC`, wat een *verlaag* operatie op een semafoor veroorzaakt.

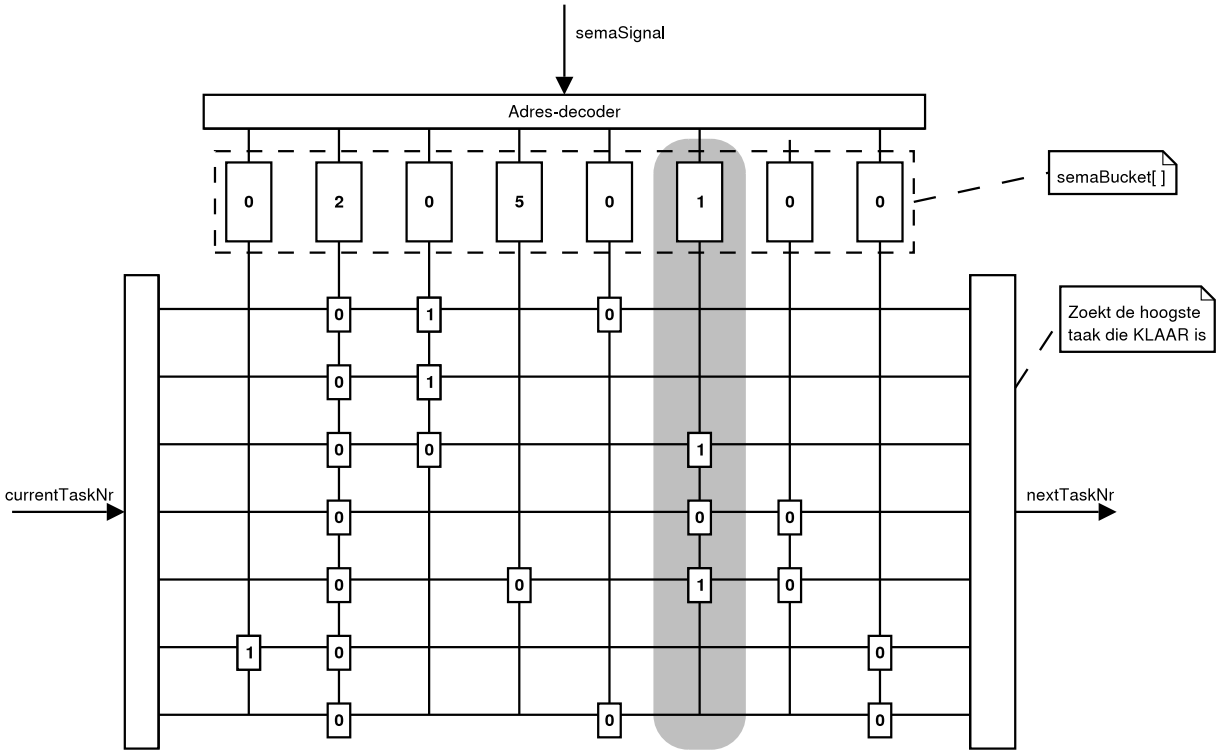
`envHandler` Intern houdt `envHandler` bij welke taak uitgevoerd wordt door de processor. `schedulerCore` beslist welke taak moet uitgevoerd worden (`nextTask` poort). Indien deze twee taken verschillend zijn, zal `envHandler` de processor onderbreken om zo een taakwissel te initiëren. De software zal aan `envHandler` vragen waar de huidige omgeving moet geschreven worden en vanwaar de nieuwe omgeving ingeladen moet worden. Om de adressen van de omgevingen te kunnen berekenen moet tijdens de initialisatie van het systeem (de *bootstrap*fase) het adres van de eerste omgeving meegedeeld worden via de poort `envBaseAddr`.

`schedulerCore`, basisimplementatie Figuur 4.5 bevat de interne stuctuur van `schedulerCore`, de kern van de scheduler. Het bevat een matrix van flipflops. De rijen stellen taken voor en de kolommen semaforen. We noemen dit de semafoor-taakmatrix. Als de waarde van een flipflop op rij r en kolom k één is, dan is taak r geblokkeerd op semafoor k. Een taak is klaar om uitgevoerd te worden als in de ganse rij van die taak enkel nullen voorkomen. De waarde van de semaforen worden opgeslagen in registers, op de figuur omkaderd met een streepjeslijn. Als een taak een bepaalde semafoor wil verlagen zal de flipflop, die overeenkomt

[2] In deze context bedoelen we met *stuurprogramma's* alle taken die reageren op signalen van hardware.



Figuur 4.5: Verfijning van `schedulerCore`.



Figuur 4.6: Geöptimaliseerde `schedulerCore`.

met die taak en semafoor, op één gezet worden. Om een semafoor te verhogen wordt de waarde van het register van die semafoor met één vermeerderd.

Beschouw elk register als een beker met knikkers. Indien het aantal knikkers (de waarde van het register) niet nul is, zal geprobeerd worden de knikker omlaag door te geven. Flipflops die de waarde nul hebben, geven de knikker verder door omlaag. De eerste flipflop die de waarde één heeft, zal de knikker bijhouden. De waarde van de flipflop wordt nul en de knikker verdwijnt. Indien de knikker niet verdwijnt bij het doorgeven, zal hij terug in de beker geplaatst worden.

`schedulerCore`, optimalisatie Afhankelijk van welke technologie wordt gebruikt voor `schedulerCore`, zal het mogelijk zijn de knikkers over verschillende flipflops door te geven in één enkele klokcyclus. Wanneer er veel taken zijn (bv. 64 taken), kan het doorgeven van de knikkers toch nog te lang duren.

Het blijkt echter dat de meeste taken maar enkele semaforen zullen gebruiken en dat de

```
void task1(void){
  for(;;){
    text=produce_text_to_be_printed();
    OsSemaWait(printer_sema);
    NOP(16); /* insert 16 NOP instructions here */
    copy_text_to_position(text,         /* protected operation */
                          next_free_position++);
    OsSemaSignal(printer_sema);
  }
}
void task2(void){
  for(;;){
    text=produce_some_text_to_be_printed(); /* but not all */
    OsSemaWait(printer_sema);
    calculate_some_more(text); /* but not too much */
    copy_text_to_position(text,   /* protected operation */
                          next_free_position++);
    OsSemaSignal(printer_sema);
  }
}
```

Figuur 4.7: Taakwissel met HWOS-2.

meeste semaforen slechts door een beperkt aantal taken worden gebruikt. De posities in de semafoor-taakmatrix die niet gebruikt worden, kunnen vervangen worden door doorverbindingen (zie figuur 4.6). Wanneer bekend is welke taken gebruik zullen maken van welke semaforen, kan men de geöptimaliseerde semafoor-taakmatrix laten genereren door een hulpprogramma. Zo'n hulpprogramma is niet geïmplementeerd.

4.2.4 Software

In het HWOS-1 zal een taak zelf een taakwissel initiëren als na een semafoorbewerking een andere taak, met hogere prioriteit, moet uitgevoerd worden. In het HWOS-2 krijgt de software geen informatie over de waarde van semaforen en kan dus niet zelf beslissen of een taakwissel moet gestart worden. Aangezien het HWOS-2 de software kan onderbreken is dit niet strikt noodzakelijk. Na een semafoorbewerking kan een taak onderbroken worden door de hardware. Naargelang de snelheid van de hardware zullen er een aantal klokcycli verstrijken tussen de se-

mafoorbewerking en de onderbreking. Tijdens deze periode zullen machine-instructies worden uitgevoerd *onafhankelijk* van het effect van de semafoorbewerking.

Deze situatie komt goed overeen met de techniek van *uitgestelde sprongen*. Bij sommige Reduced Instruction Set Computers (RISCs) zal de machine-instructie na een voorwaardelijke spronginstructie toch onvoorwaardelijk uitgevoerd worden. Men kan deze instructie nuttig werk laten verrichten, dat toch zou moeten gebeuren of dat op zijn minst niet schadelijk is voor het verdere verloop van het programma. Als alternatief kan men op deze plaats een instructie zonder effect (NOP) plaatsen.

De vertraging van het HWOS zal groter zijn dan één enkele klokcyclus. Het kan onaanvaardbaar zijn om een groot aantal NOPs uit te voeren. We zijn ons bewust van de grote programmeerinspanning die nodig is om de software te optimaliseren zodat geen NOPs moeten gebruikt worden. Figuur 4.7 bevat een taak die enkel NOPs uitvoert en een taak die nuttig werk verricht tijdens vertragingsperiode. De programmeur zal moeten kiezen tussen een tragere uitvoeringssnelheid of een moeizamer ontwikkelingstraject.

Hoofdstuk 5

Snelheid van het HWOS-2

5.1 Schatting van de snelheid

Aangezien de hardwarebeschrijving van het HWOS-2 niet synthetiseerbaar is, kunnen geen exacte voorspellingen van de snelheid van de hardware gemaakt worden. Deze snelheid zal sterk afhankelijk zijn van de gebruikte chiptechnologie, de grootte van het HWOS-2 en de optimalisaties die voorgesteld zijn. We maken een ruwe schatting voor een systeem met 64 taken en evenveel semaforen. Dit is een vrij groot besturingssysteem. Het industriële MicroC/OS-II (μC/OS-II) ondersteunt ook maar 64 taken en semaforen.

In de lijmlogica-blokken zullen maximaal enkele klokcycli gebruikt worden. We rekenen hier bijvoorbeeld op 6 cycli. Het doorgeven van de knikkers in de semafoor-taakmatrix kan 32 cycli duren, als een knikker in één klokcyclus over twee taken wordt doorgegeven en als de semafoor-taakmatrix niet geöptimaliseerd is.[1] De logica om te controleren of een taak geblokkeerd is en om te beslissen welke van de niet-geblokkeerde taken de hoogste prioriteit heeft kan in resp. 3 en 6 cycli werken.

Zo komen we op een totaal van 47 cycli voor een groot HWOS-2 zonder optimalisatie. Het valt te verwachten dat voor meer geöptimaliseerde implementaties in een snelle chiptechnologie een kortere vertraging haalbaar zal zijn.

We herhalen nogmaals dat dit een zeer ruwe schatting is.

[1] Zie de bespreking van de optimalisatie in paragraaf 4.2.3.

5.2 Meervoudige onderbrekingen

Beschouw een ingebed systeem met een aantal taken die hardware-onderbrekingen afhandelen. Deze taken kunnen vergeleken worden met stuurprogramma's in een klassiek RTOS. `driverTask` uit figuur 4.3 is een voorbeeld van zo'n taak. Ze blokkeert zichzelf meteen als ze wordt opgestart en wacht op een signaal van de hardware. Wanneer een randapparaat zo een signaal stuurt, wordt de taak gedeblokkeerd. Indien er geen taken met een hogere prioriteit kunnen worden uitgevoerd, zal een taakwissel gestart worden en de `driverTask` worden uitgevoerd.

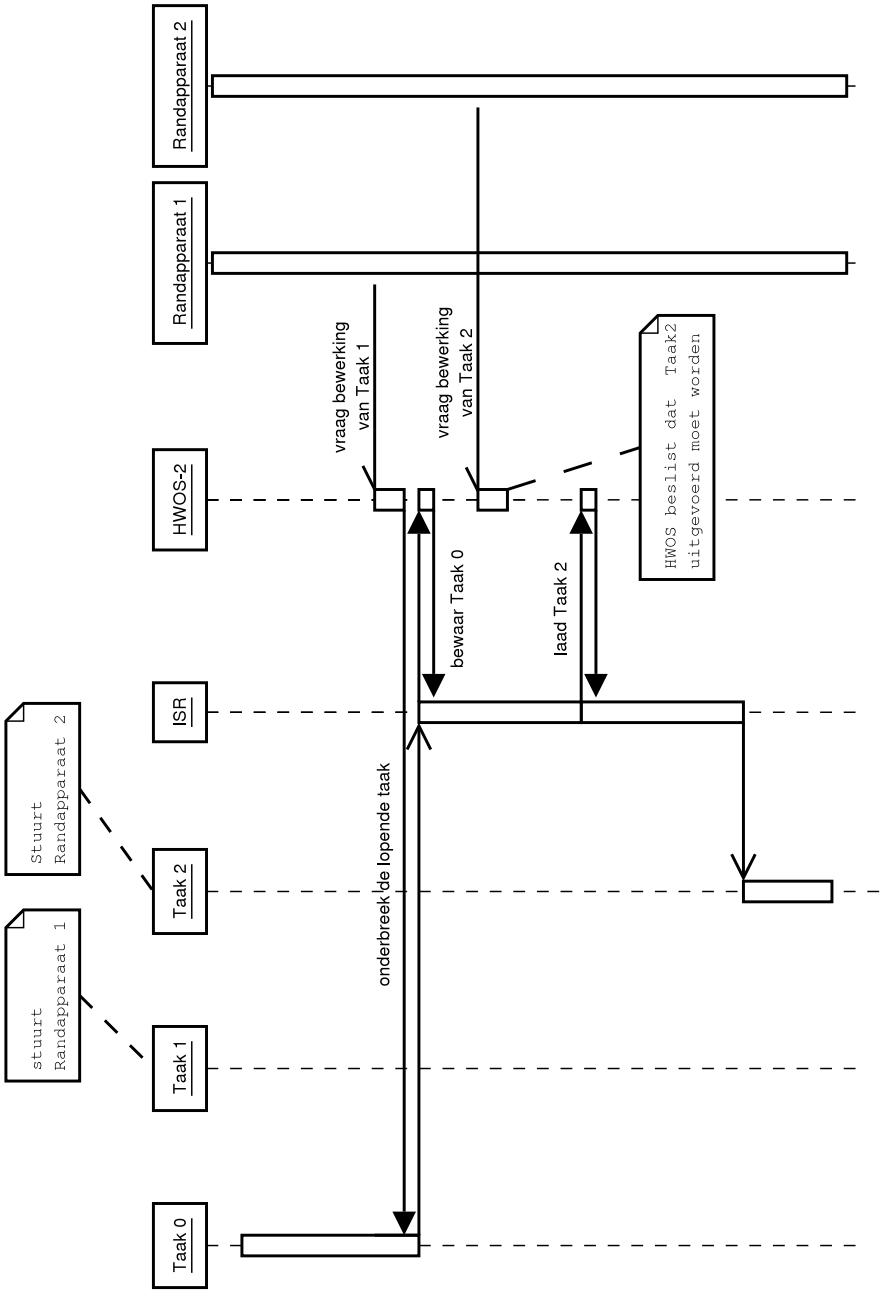
Stel dat tijdens de uitvoering van `driverTask` een andere taak wordt gedeblokkeerd door een randapparaat. Als deze taak een hogere prioriteit heeft dan `driverTask`, dan zal een taakwissel gestart worden en wordt de nieuwe taak uitgevoerd. Ook deze taak kan uiteraard onderbroken worden door een andere taak die een nog hogere prioriteit heeft.

Klassiek probeert men de onderbrekingsroutines zo kort mogelijk te houden en lange routines als gewone taken te implementeren. Dit is nodig omdat onderbrekingsroutines niet selectief kunnen onderbroken worden. Ofwel aanvaardt men alle onderbrekingen tijdens de onderbrekingsroutine, ofwel weigert men alle onderbrekingen. In het eerste geval zal de reactiesnelheid van de huidige taak verlagen als onderbrekingen met een lagere prioriteit worden ontvangen. In het tweede geval zal de reactiesnelheid van eventuele onderbrekingen met een hogere prioriteit verlagen.

Als we het HWOS gebruiken om hardwaresignalen op te vangen, is het niet nodig de onderbrekingsroutines zo kort mogelijk te maken, omdat die routines op hun beurt toch kunnen onderbroken worden zonder verlies aan reactiesnelheid. Dit gegeven zal een andere ontwerpstijl en programmeerstijl vergen voor stuurprogramma's.

5.3 Variabele tijdsduur van taakwissels

De tijdsduur van een taakwissel is niet constant, maar kan beïnvloed worden door andere taakwissels. We schetsen een situatie:



Figuur 5.1: Tijdlijn meervoudige onderbrekingen.

```
STORE:
    env=getCurrentEnv();
    store(env);
LOAD:
    env=getNextEnv();
    load(env);
if (env!=getNextEnv()) goto LOAD;
```

Figuur 5.2: Pseudocode voor verbeterde taakwissel.

Voorbeeldsituatie Beschouw de situatie van figuur 5.1, waarin `Randapparaat2` de taak `Taak2` deblokkeert, vlak nadat taak `Taak1` is gedeblokkeerd door een ander randapparaat. `Taak2` heeft een hogere prioriteit dan `Taak1`. De onderbrekingsroutine vraagt aan het HWOS waar de huidige omgeving moet geschreven worden en waar de volgende omgeving moet ingeladen worden. Als het HWOS beslist dat `Taak2` moet uitgevoerd worden *voordat* de onderbrekingsroutine de tweede vraag heeft gesteld, zal meteen de omgeving van `Taak2` ingeladen worden. Dit zal een snelheidswinst opleveren omdat de volledige taakwissel begonnen is voordat de onderbreking van `Randapparaat2` is ontvangen. Anderzijds is het ook mogelijk dat de onderbreking van `Randapparaat2` op zo'n ogenblik ontvangen wordt dat de beslissing van het HWOS om `Taak2` uit te voeren net *na* de tweede vraag van de onderbrekingsroutine komt. In dat geval zal moeten gewacht worden totdat de omgeving van `Taak1` volledig geladen is. Daarna zal de onderbrekingsroutine opnieuw gestart worden om een volledige taakwissel uit te voeren. De tijd die nodig is om te reageren op het signaal van `Randapparaat2` is dus niet constant. Er is een typische reactietijd, die voorkomt wanneer er niet net een ander signaal ontvangen is. Er is ook een maximum en een minimum reactietijd, die voorkomen wanneer de reactie op het signaal beïnvloed wordt door de afhandeling van een ander signaal. Deze situatie is niet wenselijk in een ware-tijdsysteem. De maximale uitvoeringstijd is de tijd van een taakwissel, plus de tijd die nodig is om een omgeving in te laden. De minimale uitvoeringstijd is enkel de tijd om een omgeving in te laden.

Mogelijke oplossingen Een eerste oplossing voor dit probleem bestaat erin de taakwissel zelf ook te onderbreken. Dit is mogelijk omdat op de ARM twee soorten onderbrekingen mogelijk zijn. De onderbrekingsaanvraag (IRQ) is de gewone onderbreking en de snelle onderbrekings-

Bewerkingen \ Besturingssysteem	CoWareOS	HWOS-2
Semafoorbewerking zonder taakwissel	25	8
Semafoorbewerking met taakwissel	398	90
Hardwaresignaal zonder onmiddellijke actie	~86	0
Hardwaresignaal met onmiddellijke actie	43	90

Tabel 5.1: Uitvoeringstijden

aanvraag (FIQ) is een onderbreking die voorrang heeft op de eerste soort. Er stellen zich echter problemen als een derde signaal van een randapparaat ontvangen wordt en ook de FIQ moet onderbroken worden.

Het is ook mogelijk om een aangepaste onderbrekingsroutine te schrijven die nagaat of de geladen omgeving wel de juiste is (figuur 5.2). Het inladen van een verkeerde omgeving is op zich niet schadelijk voor de verdere werking van het systeem. Men kan vlak na het laden van de omgeving van `Taak1` immers de omgeving van `Taak2` laden, zonder de omgeving van `Taak1` weer in het geheugen te schrijven. Deze laatste omgeving is immers nog niet veranderd sinds ze is geladen, want de taak heeft nog niet gelopen. Op die manier kan op een vrij eenvoudige manier de maximale uitvoeringstijd van een taakwissel sterk verkort worden. De maximale uitvoeringstijd is dan de tijd die nodig is om twee keer een omgeving te laden. Het is wel zo dat de voorgestelde aangepaste taakwissel trager zal zijn dan de originele taakwissel omdat ook extra tijd verloren wordt aan de bijkomende controle en de voorwaardelijke sprong.

5.4 Gemeten tijden

In deze paragraaf worden de gemeten tijden voor een aantal bewerkingen vermeld. De tijden in tabel 5.1 zijn uitgedrukt in aantal klokcycli en gemeten op een (gesimuleerde) ARM946 processor. We vergelijken de uitvoeringstijden van een aantal bewerkingen op een systeem dat het CoWareOS (dit is een software RTOS) gebruikt en een systeem met het HWOS-2. De tijdsvertragingen door de hardware zijn niet opgenomen in de tabel omdat we hiervoor geen exacte waarden kennen. De uitvoeringstijden voor het HWOS-2 moeten dus nog vermeerderd worden met de vertraging in de hardware. De gemeten bewerkingen zijn de volgende:

Semafoorbewerking zonder taakwissel Er wordt een `OsSemaWait` uitgevoerd op een semafoor die niet nul is.

Semafoorbewerking met taakwissel Er wordt een `OsSemaWait` uitgevoerd op een semafoor die nul is.

Hardwaresignaal zonder onmiddellijke actie Het CoWareOS ontvangt een onderbreking en start een onderbrekingsroutine die het signaal registreert en onmiddellijk terugkeert. De snelheid kan licht variëren naargelang de manier waarop het signaal wordt geregistreerd. Het HWOS-2 ontvangt een signaal van een randapparaat en registreert dit intern zonder de processor hiervoor te onderbreken.

Hardwaresignaal met onmiddellijke actie Het CoWareOS ontvangt een onderbreking, start een onderbrekingsroutine die onmiddellijk een reactie op het signaal produceert.
Het HWOS-2 ontvangt een signaal van een randapparaat en initieert een taakwissel. De nieuwe opgeroepen taak produceert een reactie op het signaal.

Als de vertraging van het HWOS-2 nuttig kan gebruikt worden en dus niet in de tijdsbalans opgenomen moet worden, is het HWOS-2 voor bijna alle beweringen sneller. De enige bewerking die trager is, is het opvangen van een hardwaresignaal met een onmiddellijke actie tot gevolg. Het CoWareOS zal deze actie implementeren als een onderbrekingsroutine. Hiertoe zullen slechts een beperkt aantal registers worden opgeslagen. De processorarchitectuur is ook geöptimaliseerd om deze bewerking efficiënt te ondersteunen. Bij het HWOS-2 zal een volwaardige taakwissel worden uitgevoerd, en dat zal langer duren. Deze taakwissel kan eventueel verkort worden door de optimalisatie beschreven in paragraaf 6.1.2.

Als de vertraging van het HWOS-2 niet nuttig gebruikt wordt, zal ook de semafoorbewerking zonder taakwissel trager zijn op het HWOS-2. Merk op dat dit voor vele van de semafoorbewerkingen niet zo erg is. Als een stuurprogramma na zijn werk zichzelf blokkeert, is het meestal niet zo belangrijk dat dit wat langer duurt. Bij het verhogen van een semafoor hoeft men ook geen NOPs uit te voeren. Men zal immers geen beschermde bewerkingen uitvoeren na het verhogen van een semafoor.

Hoofdstuk 6

Mogelijke toekomstige uitbreidingen

In dit hoofdstuk worden enkele mogelijke uitbreidingen van het HWOS opgesomd. De niet-geïmplementeerde uitbreidingen, die reeds besproken zijn in eerdere hoofdstukken, worden niet meer uiteengezet. We sommen ze hier enkel kort op:

- Processor met dubbele registers (paragraaf 2.2.1)
- Verbeterde onderbrekingsroutine voor taakwissel (paragraaf 5.3)

6.1 Specifieke HWOS uitbreidingen

6.1.1 Ondersteuning voor multiprocessors

Op multiprocessors veroorzaakt de communicatie tussen twee processors extra vertraging. Het moet mogelijk zijn om het HWOS uit te breiden zodat meerdere processors kunnen worden ondersteund. Het kan onderzocht worden of op verschillende processors een stuurprogramma geplaatst kan worden voor hetzelfde randapparaat. Wanneer het randapparaat een signaal stuurt, kan het HWOS kiezen welke processor deze aanvraag zal afhandelen.

6.1.2 Versnelde taakwissels

Taakwissels zijn vrij dure operaties. Er moeten instructies uit het hoofdgeheugen worden ingeladen, alle registers moeten worden weggeschreven naar het geheugen en nieuwe waarden van

de registers moeten worden ingeladen. Het is te verwachten dat vele taken niet alle registers zullen gebruiken. In dat geval is het niet strikt nodig dat alle registers worden opgeslagen in de omgeving van een taak. Klassiek zal men toch alle registers opslaan omdat het ook tijd zou kosten om te beslissen welke registers wel en niet opgeslagen dienen te worden. Deze vertraging zou groter zijn dan de tijdswinst die gehaald kan worden door sommige registers niet op te slaan.

Bij een HWOS kan de hardware gebruikt worden om te beslissen welke registers worden opgeslagen. Een eenvoudig algoritme kan eruit bestaan om steeds de registers op te slaan, die door de vorige taak gebruikt zijn en de registers in te laden die door de volgende taak gebruikt zullen worden. Dit algoritme is suboptimaal. Stel dat taak A alle registers gebruikt en taak B slechts twee registers. Als de twee taken alternerend worden uitgevoerd (A,B,A,B,...) zal het overbodig zijn om telkens *alle* registers van A op te slaan en weer te laden. Een complexer algoritme zou hiermee ook rekening kunnen houden.

Het zou nuttig zijn als we de compiler de instructies zouden kunnen geven om slechts een beperkt aantal registers te gebruiken voor een bepaalde taak. Zo zouden sommige taken hun werk misschien met minder registers kunnen doen. Het opleggen van een beperking in het gebruik van registers zal de uitvoeringssnelheid uiteraard nadelig beïnvloeden.

We stellen voor dat het HWOS de machinecode voor taakwissels zelf genereert op het moment van de taakwissel. Een poort van het HWOS kan afgebeeld worden op een geheugengebied. Een adres in de onderbrekingsvector zal verwijzen naar het eerste adres van dit geheugengebied. De processor zal de machinecode voor de taakwissel gewoon uitvoeren. We zijn er ons van bewust dat de overdraagbaarheid van het HWOS tussen verschillende processors bemoeilijkt. Toch lijkt ons dit de meest efficiënte implementatievorm.

6.2 Uitbreidingen uit andere RTOS'en

6.2.1 Overlopende semaforen

Zoals beschreven in paragraaf 4.2.1 zal de software niet gewaarschuwd worden door het HWOS-2 wanneer een semafoor overloopt. Als dit gebeurt zal het systeem onvoorspelbaar

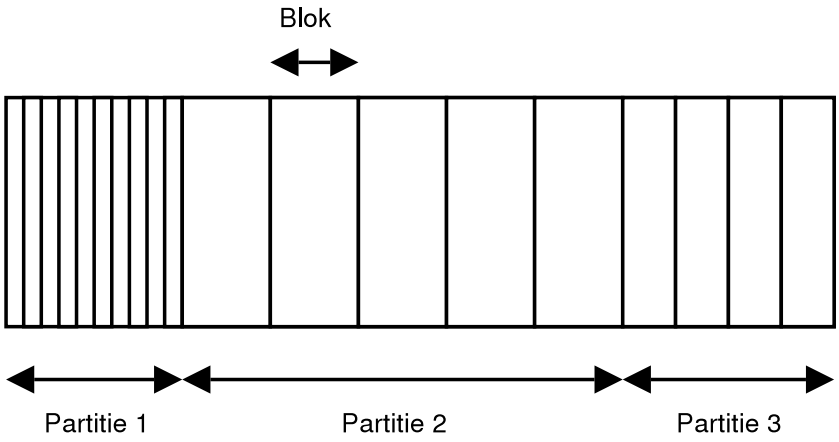
gedrag vertonen. Semaforen zouden tijdens de normale werking van het systeem nooit mogen overlopen. De systeemontwerper moet een goede inschatting maken van de maximale waarde van de semaforen en moet die waarden opgeven bij de synthese van het HWOS. Indien er toch een semafoor overloopt (dit zou door een RTOS gemeld worden), kan de software waarschijnlijk weinig anders doen dan het systeem zo veilig mogelijk afsluiten en eventueel de gebruiker waarschuwen.

Noodprocedure Het huidige HWOS-2 laat de softwareprogrammeur geen keuze aangezien de software geen gegevens heeft over de waarde van de semaforen. Het HWOS-2 zou kunnen worden uitgebreid zodat overlopende semaforen worden afgehandeld. Indien zich een overloop voordoet, detecteert de hardware dit en start een noodprocedure. Er wordt een onderbreking gestuurd naar de processor, die een taakwissel uit zal voeren. De taak die gestart wordt heeft als doel om de gebruiker te waarschuwen en het systeem veilig stil te leggen. Dit kan inhouden dat kranen en kleppen worden gesloten, verwarmingselementen, motors en zenders worden uitgeschakeld, enz. Nadat dit gebeurd is, zal de taak de werking van de processor beëindigen of, indien hiervoor geen voorzieningen zijn, een oneindige lus starten zodat het systeem niets meer kan doen.

Deze noodprocedure kan eventueel ook door de hardware worden opgestart als zich andere problemen voordoen, bijvoorbeeld een snelle daling van de voedingsspanning.

Andere semafooroperaties Bij een RTOS in software worden vaak extra bewerkingen op semaforen ondersteund. Zo is het vaak mogelijk om de waarde van een semafoor te lezen. Een uitbreiding in deze zin is mogelijk voor beide versies van het HWOS, met uiteraard een complexere hardware als gevolg.

Bij sommige RTOSen kan men proberen een semafoor te verlagen zonder het risico te lopen de taak te blokkeren. Indien de semafoor al waarde nul had, wordt de taak gewaarschuwd dat de verlaag-operatie mislukt is, maar kan de taak gewoon verder blijven lopen. Bij het HWOS-2 zou zo een uitbreiding vrij complex zijn. Het HWOS-1 kan, vanwege zijn architectuur, deze functie ondersteunen meet een relatief kleine aanpassing.



Figuur 6.1: Geheugenpartities.

6.2.2 Geheugenbeheer

Sommige RTOSen , bijvoorbeeld μC/OS-II van Jean J. Labrosse (zie [Lab02]), bieden een alternatief aan voor geheugenbeheer. Traditioneel zullen de ISO C functie `malloc` en `free` gebruikt worden om geheugen aan te vragen en weer vrij te geven. Labrosse redeneert dat deze functies niet betrouwbaar genoeg zijn voor ware-tijdsytemen. Vooreerst blijkt dat vele implementaties door het gebruikte zoekalgoritme geen constante uitvoeringstijd hebben. Dit maakt het schatten van tijdslimieten nagenoeg onmogelijk. Bovendien kan het geheugen na verloop van tijd gefragmenteerd geraken. Men bedoelt hiermee dat de vrije geheugenplaatsen in kleine hoeveelheden verspreid liggen over het ganse geheugenbereik. Het aanvragen van een groot aaneengesloten geheugenblok zal onmogelijk zijn, hoewel de totale hoeveelheid vrij geheugen groter is dan de grootte van het gevraagde blok.

μC/OS-II biedt de mogelijkheid om bij de initialisatie van het systeem een stuk geheugen te reserveren. Dit geheugen zal ingedeeld worden in een aantal partities, bestaande uit geheugenblokken van gelijke grootte. Het aantal en de grootte van de blokken worden uiteraard door de ingebedde-softwareprogrammeur bepaald. Tijdens de werking van het systeem kunnen taken blokken aanvragen. Zolang een bepaalde partitie niet leeg is, zullen blokken uit die partitie kunnen worden aangevraagd. De uitvoeringstijd van de functies om geheugen aan te vragen en vrij te geven is constant.

Dit systeem van geheugenallocatie is ook mogelijk in een HWOS. Er zal vooral snelheidswinst zijn bij een multiprocessorsysteem met gedeeld geheugen. Het toekennen van geheugen aan taken op deze of gene processor zal centraal en onafhankelijk van de software gebeuren.

6.2.3 FIFO's

Buiten semaforen is het ook mogelijk om andere IPC primitieven te implementeren in hardware. Zo is er een implementatie van First-In-First-Out geheugen (FIFO) buffers gemaakt door De Ceuster en Nieuwenhuyse [DN98]. Voor meer informatie verwijzen we naar dit werk.

Hoofdstuk 7

Conclusie

Een scheduler in hardware implementeren is een nieuw concept. Dankzij betere ontwerpprogramma's en ontwerptalen kunnen digitale schakelingen sneller en goedkoper ontworpen worden. Er bestaan reeds verschillende hulpprogramma's die digitale componenten kunnen synthetiseren aan de hand van ingevoerde parameters (bv. geheugens, optellers,...). Het HWOS kan in de toekomst misschien ook tot die lijst van synthetiseerbare componenten worden toegevoegd. Hoewel de mogelijkheden op dit ogenblik beperkt zijn, kan een HWOS zijn nut zeker bewijzen bij FPGA-gebaseerde systemen, aangezien dan ook het HWOS kan worden geüpdate. Bij systemen waar de snelheid van de processor kritiek is, kan het HWOS helpen om de processor efficiënter te gebruiken dan met een software-RTOS mogelijk zou zijn.

Er is nog onderzoek te verrichten voordat het HWOS commercieel bruikbaar is. Zo moet het HWOS nog verfijnd worden tot synthetiseerbare code en kan eventueel een generatorhulpprogramma geschreven worden. Het zou vanuit het oogpunt van CoWare ook aantrekkelijk zijn als het HWOS geïntegreerd kan worden in de N2C hulpprogramma's. Binnen het bestek van deze scriptie was dit helaas niet haalbaar.

Bijlage A

HWOS–API

Deze bijlage beschrijft de Application Programmers Interface (API) van het HWOS. Hiermee wordt bedoeld dat alle functies, gegevensstructuren e.d. die de systeemprogrammeur kan gebruiken opgenomen zijn in deze bijlage. De functies voor intern gebruik door de HWOS-software worden hier niet vermeld.

A.1 Gegevenstypes

A.1.1 `task`

Definitie

```
typedef struct {
  TaskFunction startPoint;
  int stackSize;
} task;
```

Beschrijving De gegevensstructuur `task` bevat de informatie die nodig is om een taak op te starten. `stackSize` geeft de grootte van de stapel van de taak aan in bytes. Wanneer de stapel overloopt zal het systeem onvoorspelbaar gedrag vertonen. Het is de verantwoordelijkheid van de softwareontwerper om vooraf een goede inschatting te maken van de benodigde grootte van de stapel. Dit kan eventueel gebeuren door simulaties. Labrosse stelt in [Lab02] voor om

de stapel eerst te vullen met een herkenbaar patroon en na de uitvoering van een simulatie te onderzoeken in hoeverre dit patroon overschreven is. Het gebruik van recursieve functies is ten sterkste af te raden tenzij er een maximale recursiediepte gegarandeerd kan worden.

`startPoint` is de functie die opgeroepen wordt wanneer de taak gestart is.

Voorbeeld

```
const task mytasks[]={
  {idleTask,0x50}
  {driverTask,0x1000},
  {displayTask},0x500},
  {printerTask,0x1000},
  {calculationTask,0x2000}
}
```

A.1.2 `jmp_buf_s`

Definitie

```
typedef struct {
  jmp_buf jb;
  char padding[JMP_BUF_SIZE-sizeof(jmp_buf)];
} jmp_buf_s;
```

A.2 Functieprototypes en functies

A.2.1 `TaskFunction`

Definitie

```
typedef void (*TaskFunction)(void);
```

Beschrijving In de beschrijving van een taak (zie `task`-gegevensstructuur) wordt voor elke taak een functie opgegeven waarmee de taak gestart moet worden. Deze functies zijn van het type `TaskFunction`. Ze krijgen geen parameters en mogen nooit terugkeren.

Voorbeeld

```
void myTask(void){
  initialize_my_task();
  for(;;){ /*forever*/
    keep_working();
  }
  assert_not_reached(); /*this function is never called*/
}
```

A.2.2 `OsStartTask`

Definitie

```
void OsStartTasks(const task [ ]);
```

Beschrijving Deze functie start alle taken van het HWOS op. Dit impliceert onder andere het schrijven van de omgevingen (van het gegevenstype `jmp_buf`) van alle taken en het reserveren van geheugen voor de stapels van de taken. De beschrijving van de taken staat in de rij, die als parameter wordt gegeven aan `OsStartTask`. Het aantal taken dat in die rij staat, is gedefinieerd in de (globale) preprocessormacro TASK_COUNT.

Voorbeeld

```
main(){
  OsStartTasks(mytasks);
  assert_not_reached(); /*this function is never called*/
}
```

A.2.3 `OsSemaSignal`

Definitie

```
void OsSemaSignal(int semanr);
```

A.2.4 `OsSemaWait`

Definitie

```
void OsSemaWait(int semanr);
```

Bijlage B

Implementatie van `setjmp` en `longjmp`

Deze bijlage bevat drie routines, geprogrammeerd in assemblercode voor ARM. De noodzaak en betekenis van deze routines is uiteengezet in paragraaf 3.4.

`p_nIRQ_ISR` voert een taakwissel uit.

`hwos_setjmp` is een herimplementatie van `setjmp`.

`hwos_longjmp` is een herimplementatie van `longjmp`.

B.1 `p_nIRQ_ISR`

```
p_nIRQ_ISR PROC
        sub     lr,lr,#4        ; construct return address
        mov     sp,#$currentEnv
        ldr     sp,[sp,#0]
        stmib   sp,{r0-r14}^    ; leave first position open
                                ; and store user mode r0-r14
        str     lr,[sp,#16*4]
        mrs     r0,SPSR         ; copy spsr_IRQ to r14(=lr)
        str     r0,[sp,#0]      ; stack CPSR_USR
        MOV     sp,#$interruptToCPU
```

```
        str     sp,[sp,#0]      ; notify interrupt encoder
        MOV     sp,#$nextEnv
        ldr     sp,[sp,#0]
        ldmia   sp!,{r14}       ; load CPSR_USR
        msr     spsr_cf,r14     ;
        ldmia   sp,{r0-r15}^    ; load r0_usr-r14_usr
        ENDP
```

B.2 hwos_setjmp

```
hwos_setjmp PROC                ; cf setjmp
        mov     r1,#0           ; write value 0 on unused locations
        stmia   r0!,{r1}        ; placeholder for cpsr
        mov     r1,#1           ; return value for r1 must be !=0
        stmia   r0!,{r0-r14}
        stmia   r0,{lr}         ; location for pc
        mov     r0,#0           ; return value = 0
        mov     pc,r14
        ENDP
```

B.3 hwos_longjmp

```
hwos_longjmp PROC
        ldmia   r0!,{r1}   b     ; placeholder for cpsr
        ldmia   r0,{r0-r14}
        ldr     r14,[r0,#15*4]  ; return address
        mov     r0,#1           ; return value for r0 must be !=0
        mov     pc,r14
        ENDP
```

Bijlage C

CD-ROM

Bij dit scriptieboek behoort een CD-ROM, die de volgende bestanden bevat:

Hardwaremodellen: alle modules, geschreven in CoWareC en de automatisch gegenereerde vertaling in SystemC.

Software: C programmatekst en assemblercode.

Scriptieboek: het bronbestand (in LyX-formaat) van het scriptieboek en enkele automatisch gegenereerde afgeleide bestanden (LaTeX en `pdf` formaat).

Figuren: de figuren die gebruikt werden in het scriptieboek en in de presentatie, in `dia`-formaat en in enkele andere formaten (`eps`, `png`).

Sommige `dia`-bestanden zullen momenteel niet met de officiële distributie van *dia* [dia] gelezen kunnen worden, omdat een aangepaste versie van het programma gebruikt werd om ze te creëren. De aanpassingen aan *dia* zullen door de auteur worden vrijgegeven onder de *GNU General Public License (GPL)*, maar mogelijk zal het bestandsformaat nog wijzigingen ondergaan. Daardoor kan het zijn dat geen enkele toekomstige officiële versie van *dia* de figuren van deze scriptie kan lezen. Aangezien alle figuren ook in andere bestandsformaten beschikbaar zijn, mag dit geen probleem vormen.

Presentatie De presentatie in OpenOffice.org-formaat (`sxi`-formaat) en vertaald in `ppt` formaat.

Deze bestanden maken deel uit van de scriptie en worden beschermd door het auteursrecht.

Bibliografie

[ads00a] *Assembler guide*, 2000. ARM development suite version 1.1.

[ads00b] *Developers guide*, 2000. ARM development suite version 1.1.

[ads00c] *Users guide*, 2000. ARM development suite version 1.1.

[aji] aJile Systems webpage. `http://www.ajile.com`.

[BW97] Allan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison–Wesley, 1997.

[cwr02a] *CoWare Glossary*, May 2002. Version 3.3.

[cwr02b] *CoWare N2C Design Entry Manual*, May 2002. Version 3.3.

[cwr02c] *CoWare N2C Design Execution Manual*, May 2002. Version 3.3.

[cwr02d] *CoWare N2C Design Processing Manual*, May 2002. Version 3.3.

[cwr02e] *CoWare N2C Installation Manual*, May 2002. Version 3.3.

[cwr02f] *CoWare N2C Methodology Manual*, May 2002. Version 3.3.

[cwr02g] *CoWare N2C Release Notes*, May 2002. Version 3.3.

[cwr02h] *CoWare N2C Scenario Library*, May 2002. Version 3.3.

[cwr02i] *CoWare Processor Kit Installation Manual*, May 2002. Version 3.3.

[cwr02j] *CoWare Processor Kit Manual*, May 2002. Version 3.3.

[cwr02k] *CoWare Processor Kit Release Notes*, May 2002. Version 3.3.

[cwr02l] *CoWareC Language Manual*, May 2002. Version 3.3.

[cwr02m] *CoWareOS API Manual*, May 2002. Version 3.3.

[cwr02n] *Learning CoWare N2C with CoWareC*, May 2002. Version 3.3.

[cwr02o] *Learning CoWare N2C with SystemC*, May 2002. Version 3.3.

[cwr02p] *SystemC in the CoWare N2C Environment*, May 2002. Version 3.3.

[ddd] GDB webpage. `http://www.gnu.org/software/gdb`.

[dia] Dia webpage. `http://www.lysator.liu.se/~alla/dia`.

[DN98] J De Ceuster and G. Nieuwenhuyse. Efficiente communicatiemethodes in de coware omgeving, 1998.

[gdb] DDD webpage. `http://www.gnu.org/software/ddd/`.

[gli] GLIBC webpage. `http://www.gnu.org/software/libc/libc.html`.

[gnu] GNU webpage. `http://www.gnu.org`.

[KD99] Helmut Kopla and Patrick W. Daly. *A Guide to LaTeX*. Addison–Wesley, third edition edition, 1999.

[KP98] Al Kelley and Ira Pohl. *A Book On C*. Addison–Wesley, fouth edition, 1998. Programming in C.

[Lab02] Jean J. Labrosse. *MicroC/OS-II*. CMP Books, second edition, 2002. the real-time kernel.

[lyx] LyX webpage. `http://www.lyx.org`.

[Sim99] David E. Simon. *An Embedded Software Primer*. Addison–Wesley, 1999.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, third edition, 1997.

[sys02] *SystemC version 2.0 User's Guide*. www.systemc.org, 2002. Update for SystemC 2.0.1.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition edition, 2001.

[You82] S. Young. *Real Time Languages: Design and Development*. Ellis Horwood, 1982.