Virtual eXecuter

Indhold

[Indledning 2](#_Toc198657249)

[Systemoversigt 3](#_Toc198657250)

[Systembeskrivelse 3](#_Toc198657251)

[Systemarkitektur 3](#_Toc198657252)

[Den virtuelle maskines arkitektur 4](#_Toc198657253)

[Processlisten 4](#_Toc198657254)

[Instruktionsfortolkning 5](#_Toc198657255)

[Periferienheder 5](#_Toc198657256)

[Øvrigt 6](#_Toc198657257)

[VX programmer 6](#_Toc198657258)

[Minimal Resource Operating System 6](#_Toc198657259)

[Hardwareplatform 7](#_Toc198657260)

[Værktøjer 8](#_Toc198657261)

[Virtual eXecuter Image Creator 8](#_Toc198657262)

[Virtual eXecuter Terminal 8](#_Toc198657263)

[Firmware Updater 8](#_Toc198657264)

[Designbeslutninger 8](#_Toc198657265)

[Afvikling fra RAM 8](#_Toc198657266)

[Anvendelse af DRAM 8](#_Toc198657267)

[Test resultater 9](#_Toc198657268)

[Første tests 9](#_Toc198657269)

[Et par optimeringer senere… 9](#_Toc198657270)

[Konklusion 9](#_Toc198657271)

[Bilag 10](#_Toc198657272)

[VX instruktionssættet (uddrag fra VX manualen) 10](#_Toc198657273)

[Todo 10](#_Toc198657274)

# Indledning

Målet med dette projekt var at opbygge en virtuel maskine der kunne afvikles på en AVR microcontroller. Det vil altså sige at der skulle defineres og implementeres en komplet processor i software.  
En VM i sig selv er naturligvis ikke speciel interessant hvis ikke det er muligt at tilgå og afvikle programmer på den så projektet inkluderede også et filsystem hvor applikationer kan gemmes og et terminal interface så det er muligt at styre systemet.

# Systemoversigt

Logisk set består systemet af 4 dele som vist herunder



Et mere detaljeret billede afslører komponenterne i de 4 dele



# Systembeskrivelse

Block diagrams & figures

Define system architecture & user interface

# Systemarkitektur

Internt består systemet groft set af tre dele

VX_Terminal_MROS.emf

Systemet er baseret på mit egetudviklede kooperative operativ system MROS (Minimal Ressource Operating System). MROS tilbyder en time base for systemet, fuld bufferet og interrupt drevet serielkommunikation, trådhåndtering, dynamisk hukommelsesallokering og en række hjælpefunktioner til blandt andet strenghåndtering.  
Den virtuelle maskine består af et antal moduler der afvikles som en tråd på MROS. Denne tråd oprettes ved system opstart.  
Gennem terminalen er det muligt at anvende systemet på samme måde som det kendes fra fx D OS og Linux.

# Den virtuelle maskines arkitektur

Den virtuelle maskine består af en række moduler hver med sit ansvarsområde. Disse er

* Processliste manipulation og inspektion
* Instruktionsfortolker
* Periferienheder
* Øvrige maskinrelevante handlinger

Herunder beskrives de enkelte dele i maskinen, hvad de gør og hvordan de virker.

## Processlisten og processhåndtering

Alt i VMen tager udgangspunkt i processlisten. Denne indeholder de processor som brugeren har oprettet (fra terminalen). Rent fysisk er processlisten lokaliseret i DRAMen som en linked list.

Når en ny process oprettes reserveres den nødvendige plads og denne plads opdeles i fire segmenter.



Meta segmentet indeholder følgende information

* Id. Hver proces har et unikt id så det er muligt at pege på en vilkårlig proces fra terminalen. Rent praktisk svarer dette id til den adresse i DRAMen så processen starter på. Dette gør det enkelt at finde et unikt id ved processoprettelse da adressen jo af natur er unik.Samtidig gør det det nemt at oversætte id til adresse og omvendt.
* Options. Dette er de 32 bit også findes i .vxx filer. Dette felt initialiseres ved opstart når processen læses ind i hukommelsen. Disse er endnu ikke brugt til noget.
* Navn. Processens navn svarer til programfilens navn. Dette gør det nemt for brugeren at overskue kørende processer når disse udskrives i terminalen.
* Start adressen for hvert af de tre segmenter Code, Data og Stack. Disse bruges til at udregne den absolutte adresse når der peges ind i et segment. Disse udregnes ved processoprettelsen.
* Størrelsen for hvert af de tre segmenter. Disse indlæses fra programfilen ved oprettelsen og bruges til at verificere at programmet ikke er kørt uden for dets hukommelsesområde.
* Processens tilstand. Se beskrivelsen nedenfor.
* Næste proces i listen. Af optimeringshensyn next pointeren for den linkede liste som udgør processlisten integreret i hver proces’’ meta information.
* Ticks. Hver proces har en tæller der indikerer hvor mange instruktioner den har udført.
* ALU flag. Resultatet af hver handling der placerer en værdi på stakken påvirker disse flag så det er nemt at undersøge om sidste udregning fx resulterede i værdien nul. Dette bruges af jump instruktionerne.
* En pointer ind i hvert af de tre segmenter. Disse peger på den næste instruktion der skal udføres, den næste plads i stakken og starten af den nuværende stack frame ved metodekald.

En process kan være i én af fire tilstande:

* Stop
* Run
* Step
* Crash

Når en process oprettes sættes den i tilstanden Stop. Dette betyder at der ikke afvikles instruktioner.  
Tilstanden Run betyder at processen afvikles.  
Tilstanden Step anvendes til at single steppe igennem en process. I denne tilstand afvikles der kun en instruktion fra processen hvorefter den skifter til tilstanden Stop.  
Tilstanden Crash betyder at en af følgende hændelser skete:

* Processen forsøgte at afvikle en ugyldig instruktion
* Program Counter (PC) pegede uden for Code Segment da der forsøgte at blive læst herfra.
* Stack Pointer (SP) pegede uden for Stack Segment under en push operation eller der blev forsøgt at læse flere bytes fra stakken end der var pushet på den.
* Stack Frame Pointer (SFP)

Ingen af disse hændelser påvirker andre processer. Det vil fx sige at en process der forsøger at placere flere bytes på sin stak end der er allokeret til denne vil ikke kunne overskrive data fra andre processer. Alle processor arbejder i deres eget afgrænsede hukommelsesområde.  
Er en process først kommet i Crash tilstanden er det ikke muligt at skifte tilstanden for denne igen. Den eneste operation der nu kan udføres på processen er Kill.

## Instruktionsfortolkning

Instruktionsfortolkeren afvikles som en MROS tråd. Når denne tråd tildeles processortid indlæses alle processorne i processlisten en efter en og der afvikles én instruktion fra de processor hvis tilstand tillader det.

Selve fortolkningen af en enkelt instruktion sker ved at instruktionen hentes fra den plads der peges på af PC. Værdien af denne instruktion bruges herefter som indeks i instruktionstabellen og funktionen på denne plads afvikles. Peges der på en ugyldig plads i denne tabel, altså indekset for en ikke implementeret instruktion, skifter processen til tilstanden Crash.

Hver instruktion er implementeret som en C funktion i modulet ProcessExecuter. Denne funktion udfører de nødvendige push/pop instruktioner og behandler data som nødvendigt. Dette medfører en række funktionskald og da der samtidig checkes for eventuelle fejltilstande tager afviklingen af en enkelt VX instruktion adskillige processor klokcyklusser. Netop her kunne der optimeres for at gøre VX maskinen hurtigere.

Når en instruktion er blevet udført skrives processens nye tilstand tilbage i processlisten. Her er optimeret så kun de felter der ændrer sig skrives tilbage. Det vil fx sige stack pointeren. Processens navn skrives ikke tilbage da denne jo ikke ændres ved afvikling af instruktionerne.

Kun cirka 80 af de 256 mulige instruktioner er implementeret og der er altså rig mulighed for senere udvidelser. Oplagte udvidelser er trådhåndtering og dynamisk hukommelsesallokering.

## Periferienheder

Begrebet periferienheder er inspireret af den måde gængse microcontrollere kommunikerer med deres tilknyttede hardwareenheder (fx en UART) på.

VX instruktionssættet tillader læsning og skrivning af enkelte bytes på 1 af 256 input/output adresser. Disse er internt forbundet med nogle moduler kaldet software periferienheder eller SoftPeripherals.  
Disse udvikles som moduler med en, to eller tre funktioner der forbindes til VX maskinen ved at placere dem i en tabel med 256 elementer. Disse tre funktioner er Init, Read og Write.

Init implementeres kun hvis det udviklede modul kræver initialisering ved system opstart. Ikke alle moduler har behov for dette og denne funktion kan da blot undlades.

Read funktionen kaldes når en VX applikation udfører en In instruktion på den IO lokation som modulet er forbundet til. Her udfører modulet den nødvendige logik for at hente en byte fra en nødvendig hardwareenhed eller hvad der nu er relevant for det konkrete modul. Funktionen returnerer herefter værdien og SoftPeripherals modulet sørger herefter for at værdien placeres på stakken så den efterfølgende er tilgængelig for applikationen.

Write operationen udføres når VX applikationen afvikler en Out instruktion på modulet. Byten der sendes fra applikationen til modulet overføres til funktionen og kan herefter anvendes inden i denne.

SoftPeripheral systemet er meget fleksibel og tilbyder en enkel måde at udvide med nye periferienheder og kun de nødvendige operationer (Init, Read og Write) behøves implementeret.  
Hvis et modul ikke tilbyder fx Read ignoreres In instruktioner på det givne modul og der returneres blot værdien 0 (nul). Dette sikrer det forventede stakforbrug og programafviklingen kan fortsættes uden problemer andet end at en eventuel forventet handling udebliver.

## Øvrigt

Udover de beskrevne elementer er der mindre detaljer i forbindelse med den virtuelle maskine som fx initialisering af denne.

# VX programmer

Der skelnes i VX miljøet mellem programmer og processor.

* Et program er en binær fil på filsystemet der kan indlæses i hukommelsen på VX systemet.
* En process er et program der er indlæst i hukommelsen og efterfølgende tilføjet til processlisten.

Programmer har filefternavnet .vxx der står for Virtual eXecuter EXEcutable. Der ud over er der et mærke i starten af filen i form af bogstaverne VXEXE som bliver verificeret ved programindlæsning. Begge disse ting skal være opfyldt for at VXen oprette en process ud fra programmet.

Efter VXEXE mærket følger et antal felter som bruges af VX loaderen til at verificere at der plads til at oprette processen i hukommelsen og som senere bruges til at kontrollere processens integritet.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Filmærke** | **Options** | **Kodestørrelse** | **Datastørrelse** | **Stakstørrelse** | **Kode** |
| VXEXE | Reserveret | Antal bytes i kodesegmentet | Antal bytes i datasegmentet | Antal bytes i staksegmentet | Selve programmet |

Filmærket optager 5 bytes og de næste fire felter optager hver 4 bytes. Størrelsen af kodeblokken svarer til det feltet ”Kodestørrelse” angiver.  
De tre størrelsesangivelser er som alt andet i såvel VX systemet som MROS i little endian format.

Der ud over er der påtænkt et komprimeret programformat benævnt Virtual eXecuter Compressed EXecutable med filefternavnet .vxc og filmærket VXCEX. Dette er ikke implementeret.

# Minimal Resource Operating System

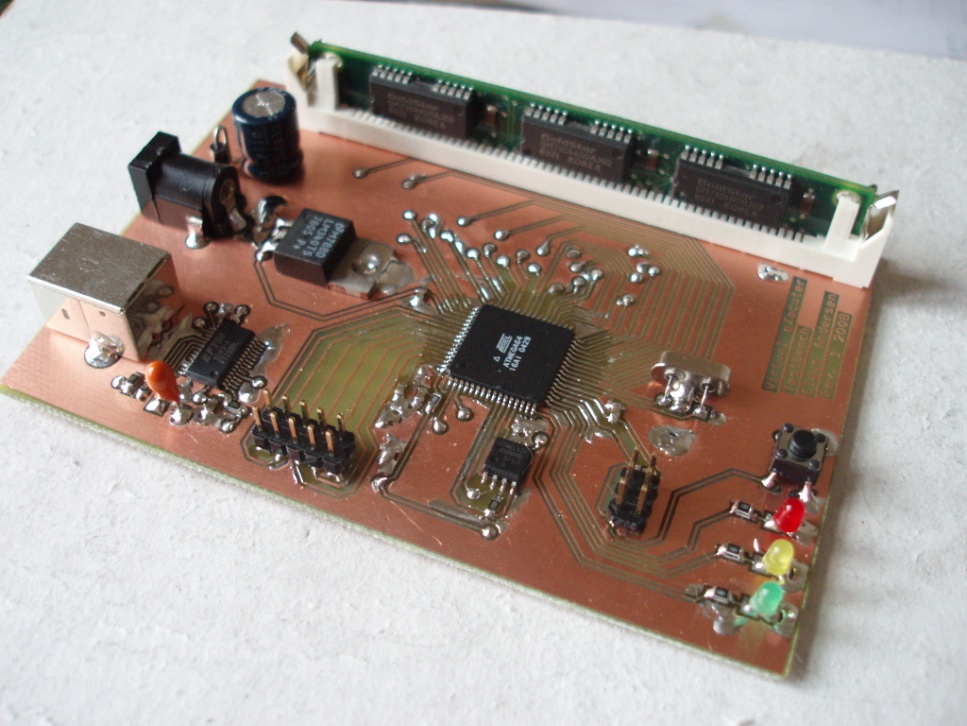
Grundlaget for hele systemet er operativsystemet MROS. Dette OS er specialdesignet til små indlejerede systemer med helt ned til 64 bytes RAM og 1 kB flash og specielt målrettet AVR processoren.  
MROS tilbyder grundlæggende trådhåndtering med tidsstyring, semaporer, message queues og mail boxes. Trådene afvikles kooperativt hvilket medfører det minimale foot print. Der ud over tilbydes en række moduler til dynamisk hukommelsesallokering, et filsystem, interrupt drevet, bufferet serielkommunikation, terminal interface og TCP/IP stak (foreløbig begrænset til IP, ICMP, ARP, og UDP). Yderligere er der moduler til operationer på tekststrenge.  
Dette er udviklet på forhånd og er som sådan ikke en del af VX projektet men er grundlaget for dette.  
MROS er ikke kompatibelt med standard C biblioteksfunktioner til fx læsning og skrivning af filer etc. og dette er et bevist valg da disse ofte har et stort foot print i forhold til en lille microcontroller. Alt hvad der måtte findes i standardbibliotekerne findes i MROS (eller bliver tilføjet efterhånden som behovet opstår) og disse biblioteker kan derfor helt undgås.

# Hardwareplatform

Til at teste VX systemet blev der fremstillet et print med følgende komponenter:

* Processor. En AVR ATmega64 med 64 kB flash, 4 kB SRAM og 2 kB EEPROM. Processoren kører med en klokfrekvens på 11,0592 MHz.
* USB interface. Der anvendes en FT232RL USB kreds fra FTDI chip til at kommunikere med en PC. Denne kreds forbinder systemet til en PC som via en klassisk seriel port. Der kommunikeres ved 460800 bps.
* DRAM sokkel med understøttelse for op til 16 MB DRAM. Her kan monteres et 30 pins DRAM modul som dem der tidligere blev anvendt i PCer. Der er lige nu monteret 1 MB.
* 8 IO ben er ført ud i et stik for at systemet kan kommunikere med eksterne hardware enheder.
* Der ud over er der monteret en 512 kB DataFlash kreds som kan bruges til filsystemet (indtil videre anvendes blot den interne EEPROM).
* Printet har sin egen strømforsyning som kan anvendes hvis der forbindes strømforbrugende eksterne enheder. Kører printet uden eksterne enheder er det nok blot at forbinde et USB kabel og printet forsynes da her igennem.

Jeg valgte hurtigt at projektet fortjente et ordentligt print frem for at basere systemet på løsdele forbundet med ledninger. Kommunikationen til specielt DRAMen er mere støjfølsom en jeg turde overlade til en samling løse ledninger og de medførende problemer det uden tvivl ville give. Det endelige resultat ses herunder.



Den grønne lysdiode viser systemets tilstand og lyser når systemet kører. Når der udføres kommandoer afsendt fra terminalen slukkes denne. De to øvrige lysdioder kan anvendes fra VX applikationer via et SoftPeripheral.

# Værktøjer

## Virtual eXecuter Image Creator

Filsystemet der er anvendes i projektet hedder FileStore og er en del af MROS. Det er udviklet specielt til meget små systemer med blot få kilobytes plads og kan derfor anvende den interne EEPROM i AVRen som kun er på 2 kB. For at gøre det så kompakt er det ikke muligt at ændre eller oprette filer når systemet kører. Derfor skal et disk image forberedes på PCen og downloades til EEPROMen. Dette gøres med VXIC der kan lave et image med filefternavnet .vxi ud fra et bibliotek. VXIC laver ukritisk et image med alle de filer (underbiblioteker er ikke understøttet) der ligger i biblioteket. Om dette image så kan være på mediet skeles der ikke til.

## Virtual eXecuter Terminal

Der er udviklet et specielt terminalprogram i forbindelse med VX projektet. Dette gjorde jeg fordi det endnu ikke var lykkedes mig at finde et simpelt terminalprogram med lige præcis de muligheder jeg ville have og ikke 1000 jeg ikke ville have.  
Resultatet blev VXT; et konsolbaseret terminalprogram der kan udføre specialkommandoer.

Den eneste specialkommando der er implementeret er ”load”. Load bruges til at downloade disk images til systemet. Load er som sådan blot en almindelig kommando i systemets terminal men protokollen der bruges til at sende filen efterfølgende kan ikke udføres fra terminalprogrammet.

## Firmware Updater

For at lette udviklingsforløbet blev der anvendt en bootloader til at opdatere firmwaren i systemet.  
Bootloaderen og den tilhørende PC applikation var som MROS udviklet på forhånd og skulle blot konfigureres til at passe til det konkrete system.

# Designbeslutninger

## Afvikling fra RAM

I praksis ville et setup som dette med en 8 bit processor med ekstern DRAM aldrig finde anvendelse. DRAM skal genopfriskes med korte mellemrum og denne memory refresh bruger 5 % af processortiden alene. Der ud over er der den øgede tilgangstid. Et mere anvendeligt virtuel maskine baseret system ville have afviklet programmet direkte fra det medie hvorpå programmerne var opbevaret. Dette var jeg fuldt ud bevidst om fra starten men valgte alligevel at undersøge dette princip for at undersøge mulighederne.

## Anvendelse af DRAM

AVRen har ikke indbygget understøttelse af dynamisk RAM og der bruges derfor relativt meget tid på at tilgå arbejdshukomelsen. Dette kunne være løst ved at anvende en statisk RAM for hvilken der er indbygget hardwareunderstøttelse. Dette ville have begrænset hukommelsesstørrelse til 64 kB inden der skulle benyttes sløvende pageing principper men tilgangstiden ville stadig have været betydelig kortere.

Jeg valgte hovedsageligt DRAM for at undersøge muligheden for at anvende en (i 8 bit verdenen) enorm størrelse RAM og for at se hvor stor tabet ville være.  
Der ud over fik jeg bedre mulighed for at anvende hukommelsestunge datastrukturer og for at prøve oprette mange større processorer.

# Test resultater

## Første tests

Da der første gang kom ”hul igennem” og de første instruktioner var blevet implementeret blev systemydelsen målt til lidt over 1500 instruktioner per sekund (ips). Dette var ikke umiddelbart imponerende men samtidig heller ikke overraskende. Hver instruktion skal inden den udføres hentes fra DRAMen og da det bestemt ikke er meningen at en AVR microcontroller skal anvende DRAM tager det en anseelig mængde tid at hente en byte.

## Et par optimeringer senere…

For at forsøge at forbedre ydelsen af systemet blev DRAM læse og skrivefunktionerne efterfølgende implementeret i assembler mod tidligere i C. Dette sparede et par clocks på begge funktioner og bragte den samlede ydelse op med lidt over 2 %. Dette demonstrerede samtidig en af de få situationer hvor det kan betale sig at implementere dele i assembler på trods af den øgede kompleksitet og den mindskede portabilitet.

Disse benchmarks blev udført på et meget snævert udsnit af instruktionssættet da kun enkelte instruktioner var implementeret da de første tests blev udført og andre testapplikationer vil give et helt noget andet resultat da der kan være stor forskel på eksekveringstiden af de enkelte instruktioner.

Der er efterfølgende ikke gjort yderligere for at optimere på ydelsen da den er af mindre betydning i denne sammenhæng.

# Konklusion

Jeg er positivt overrasket over hvor godt et resultat jeg har opnået. Takket være MROS og bootloaderen har hele udviklingsprocessen været stort set gnidningsløs og jeg har kunnet fokusere på de relevante punkter.

Det udviklede demonstrationsboard har siden tidligt i projektet givet mig en stabil platform at udvikle på så når noget ikke har virket har jeg hurtigt kunne indsnævre det til det sted i koden hvor fejlen var.

Selve den virtuelle maskine var umiddelbart nemmere at få op at køre en forventet. Ydelsen og dermed også til dels anvendeligheden var ikke imponerende men det var et bevidst valg at ofre ydelse til fordel for et lækkert design.

# Bilag

## VX instruktionssættet (uddrag fra VX manualen)

# Todo

Referencelinks.

CD: firmware + MROS. VXT. VXIC. Datasheets. Firmware Updater SW + FW. Referencedokumenter