Ingeniørhøjskolen i Århus
IKT
Dalgas Avenue 2
8000 Århus C

29. maj 2006

ITJAV2
Obligatorisk opgave
Portering af KVM til RTKernel

Studerende:
Henrik Brix Andersen, 01079
Tomas Stæhr Berg, 03539
Benjamin Hedegaard Sørensen, 02284

Underviser:
Bjarne Andersen

Indhold

1 Kravspecifikation

Dette afsnit beskriver kravene til det udviklede system ud fra laboratorieøvelse 8 og 10 i kurset Java 2. Opgaveformuleringen er suppleret med egne kommentarer samt en systemoversigt.

1.1 Formål

Formålet med laboratorieøvelse 8 og 10 er at arbejde med et SBC686-baseret RTKernel system indeholdende en KVM (K Virtual Machine). Hardwarekortet IO686 anvendes i en vis udstrækning.

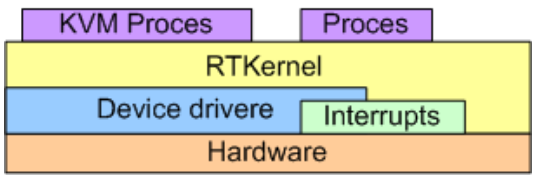
1.2 Problemformulering

Arbejdet kan inddeles i følgende opgaver:

1. Virtual Machine Windows: Bygning af Sun's referenceimplementering af J2ME CLDC(KVM) til Windows og test af denne.
2. Virtual Machine RTKernel: Portering af KVM til OnTime RTKernel systemet, oprettelse af et simpelt RTK system og test af dette.
3. RTKernel processer/opgaver:
   - Proces Main: Initierer, starter og lukker systemet ned.
   - Proces LEDPort: Viser den aktuelle tilstand af LED'erne på IO686-kortet.
   - Proces SwitchPort: Viser den aktuelle indstilling af switchen på IO686-kortet.
   - Proces Operator: Kommandofortolker.
   - Proces KVM: Starter KVM'en, holder styr på de nødvendige oplysninger og viser relevant information (main-class, classpath, fejlmeddelelser osv.)
   - Proces JavaOut: Kan vise output fra det kørende Java-programmel.
4. KNI: K Native Interfaces, der giver Java-programmer mulighed for at kalde native kode. Skal bruges til aflæsning/skrivning til hardware, output fra Java-programmet og til interrupt-håndtering.
5. Java-program: Skal kunne aflæse switchen på IO686-kortet, foretage en beregning og skrive resultatet til LED'erne. Beregningsdelen skal designes med et `IO686Calculate` interface, der giver mulighed for udskiftning af den egentlige beregningsdel. Derudover skal programmet være i stand til at lukke ned, hvis det får besked om en interrupt fra RTKernel-systemet.

1.3 Systemoversigt

Systemet er illustreret på figur 1.1. Real-Time operativsystemet RTKernel kommunikerer hardwaren med via device-drivere og interrupts. I operativsystemet kan der startes forskellige processer, f.eks. en KVM. KVM'en starter et Java program, som for operativsystemets synspunkt er en del af KVM-processen. Java programmet kan kalde andre processer i systemet via KNI. KNI-modulerne og processerne kan kommunikere med hinanden via IPC-ojekter.



Figur 1.1: Systemoversigt

2 Strukturering

Dette afsnit beskriver struktureringen af systemet. En moduloversigt kan ses på figur 2.1 på næste side. Modulerne til venstre for den stiplede linje er implementeret i programmeringssproget C ved anvendelse af RTKernel. Modulerne til højre for den stiplede linje er implementeret i Java og eksekveres under KVM'en.

Systemet eksponerer funktionaliteten af IO686-kortets DIP-switche, LED-panel og interrupt-knap via KNI til anvendelse i Java-programmel.

De enkelte moduler, deres ansvarsområder samt systemets filstruktur er beskrevet i de efterfølgende afsnit.

2.1 Device-drivere

For at lette tilgangen til IO686-kortets funktioner er der implementeret to device-drivere, som er beskrevet i de følgende afsnit. Hver driver er internt beskyttet med en semafor for at sikre, at kun én proces kan tilgå driveren ad gangen.

2.1.1 LEDport

LEDport-driveren eksponerer IO686-kortets 8 LEDs via C-funktionskald. Driveren tillader slukning og tænding af hver enkelt LED.

2.1.2 Switchport

Switchport-driveren eksponerer IO686-kortets 8 DIP-switche via C-funktionskald. Driveren tillader aflæsning af status for hver enkelt DIP-switch samt samlet aflæsning af alle 8 DIP-switche som én oktet.

2.2 Processer

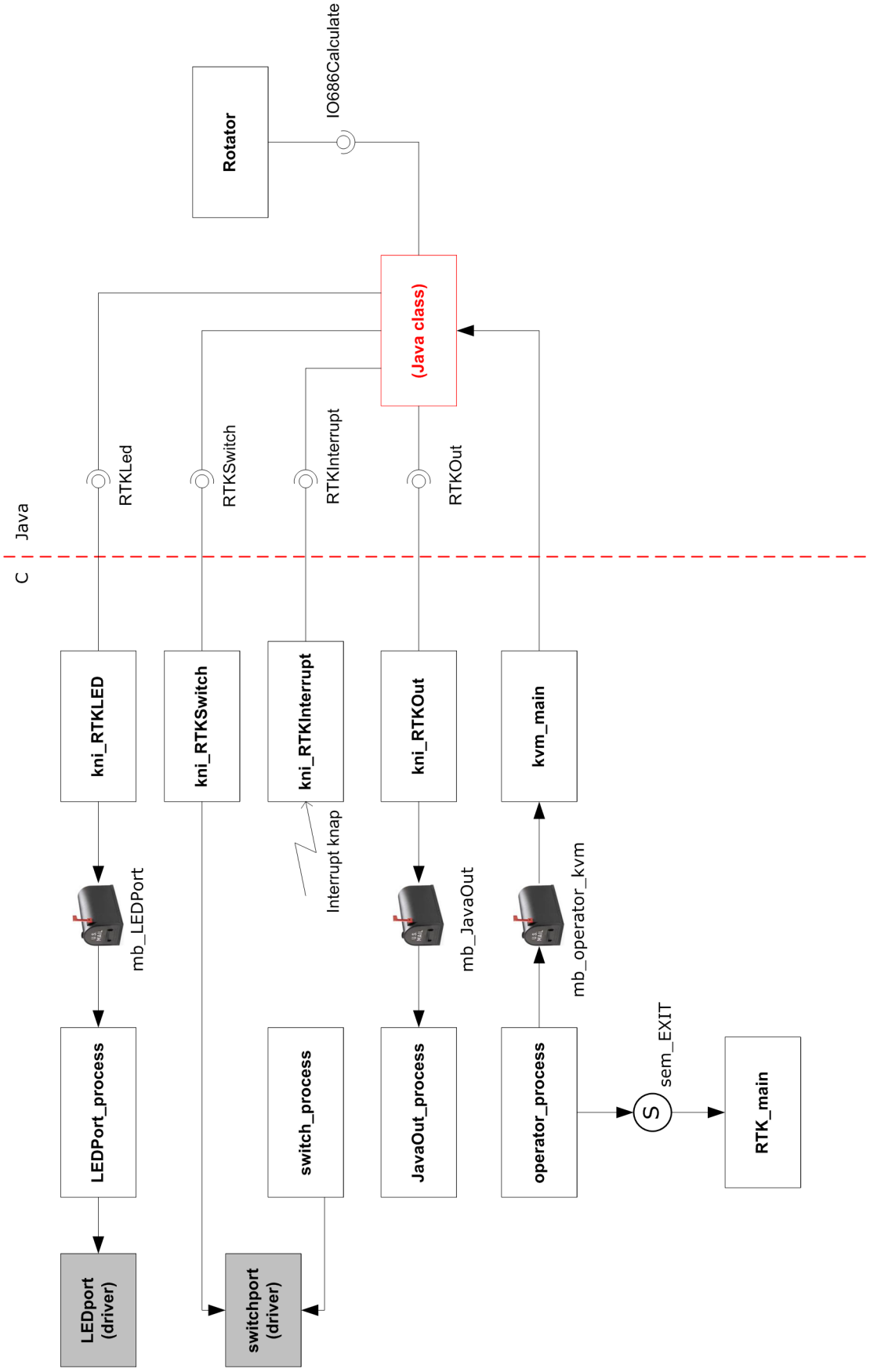
Systemet er bygget op omkring seks forskellige processer med hvert sit ansvarsområde. Hver proces har et vindue i den grafiske brugergrænseflade til udskrivning af statusbeskeder. De enkelte processer er beskrevet i de følgende afsnit.

2.2.1 LEDport_process

LEDport-processen har ansvaret for at tænde og slukke IO686-kortets LEDs via `LEDport`-driveren. Der modtages besked om ønskede ændringer igennem postkassen `mb_LEDPort`. Processen udskriver løbende LED-portens status via den grafiske brugergrænseflade.

2.2.2 switch_process

Switch-processen har til opgave at polle IO686-kortets DIP-switche via `switchport`-driverent, og udskrive deres tilstand til den grafiske brugergrænseflade.



Figur 2.1: Moduloversigt

2.2.3 JavaOut_process

JavaOut-processen har til opgave at udskrive tekst-beskeder modtaget via postkassen `mb_JavaOut` til den grafiske brugergrænseflade.

2.2.4 operator_process

Operatorprocessen har anvaret for indlæsning og fortolkning af brugerinput.

- Indlæses et 'x' signaleres `RTK_main`-processen igennem semaforen `sem_EXIT` og hele systemet lukkes ned.
- Indlæses et 'c' kan brugeren indtaste en classpath. Denne sendes til `KVM_main`-processen via postkassen `mb_operator_kvm`.
- Indlæses et 'k' kan brugeren indtaste navnet på en Java-fil, som skal eksekveres. Filnavnet sendes til `KVM_main`-processen via postkassen `mb_operator`.

2.2.5 kvm_main

Denne proces har ansvaret for at starte KVM'en med parametrene modtaget via postkassen `mb_operator_kvm`.

2.2.6 RTK_main

Denne proces er systemets hovedprogramløkke. Den har ansvaret for at starte de øvrige processer i systemet samt at initialisere IO686-driverne.

Ved signalering fra semaforen `sem_EXIT` afbrydes hovedprogramløkken og systemet lukkes ned.

2.3 K Native Interfaces

Systemet eksponerer fem K Native Interfaces (KNI) til anvendelse fra Java-programmellet. Disse fem interfaces er beskrevet i de følgende afsnit.

2.3.1 RTKLed

Interfacet `RTKLed` er stillet til rådighed af modulet `kni_RTKLED`. Dette interface tillader tænding og slukning af IO686-kortets LEDs. Modulet kommunikerer med `LEDPort`-processen via postkassen `mb_LEDPort`.

2.3.2 RTKSwitch

Interfacet `RTKSwitch` er stillet til rådighed af modulet `kni_RTKSwitch`. Dette interface giver mulighed for aflæsning af IO686-kortets DIP-switche. Modulet kommunikerer direkte med `switchport`-driveren.

2.3.3 RTKInterrupt

Interfacet `RTKInterrupt` er stillet til rådighed af modulet `kni_RTKInterrupt`. Interfacet giver mulighed for at undersøge, om IO686-kortets interrupt-knap har været aktiveret. Modulet implementerer sin egen RTKernel interrupt-handler, der kaldes ved aktivering af interrupt-knappen.

2.3.4 RTKOut

Interfacet `RTKOut` er stillet til rådighed af modulet `kni_RTKOut`. Dette interface giver mulighed for udskrivning til RTKernels brugergrænseflade fra Java-programmellet.

2.4 Java-interfaces

Til beregning på værdierne aflæst på IO686-kortets DIP-switche er der implementeret et Java-interface, som beskrevet i det følgende afsnit.

2.4.1 IO686Calculate

Interfacet `IO686Calculate` er implementeret i Java-klassen `Rotator`. Denne klasse giver mulighed for bit-vis rotering af aflæste data. Anvendelsen af et interface giver mulighed for implementering af flere beregningsfunktioner implementeret i andre klasser. Det er ikke muligt at udskifte beregningsdelen dynamisk under KVM, da klasser ikke kan loades på runtime, se 5.1.2 på side 10.

2.5 Filstruktur

Dette afsnit beskriver mappestrukturen og filerne i vores KVM-kildetekst.

Mapper	
api	J2ME API klasserne, som kan kaldes fra et Java-program.
bin	Prekompileret kvm og preverifier tool.
kvm	C filer, der implementerer KVM og API'et, og Visual Studio projekter til at bygge KVM.
tools	Kildekoden til hjælpeværktøjerne jcc, preverifier og kdp.
Filer	
build.bat	Batchfil som bygger API'et.
RTK_filelist	Liste af alle java filerne, der skal kompileres.
RTK_options	Liste af parameter til kompileringen af API'et
make.bat	Batchfil som kompilerer og preverifier en java fil udfra KVM API'et.
zip.exe	Pakkeprogram.

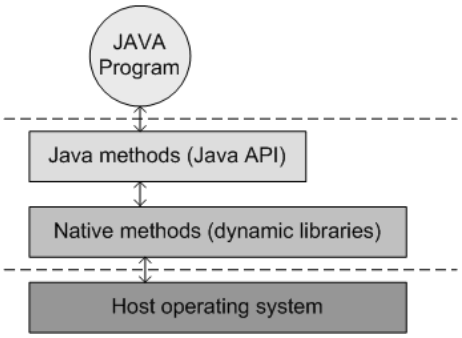
Tabel 2.1: Filstruktur

3 Implementering

Dette afsnit beskriver implementeringen. Vi ser på hvordan KVM'en bygges, hvordan vi starter en KVM fra et RTKernel-program og hvordan KVM'en kan interagere med RTKernel operativsystemet.

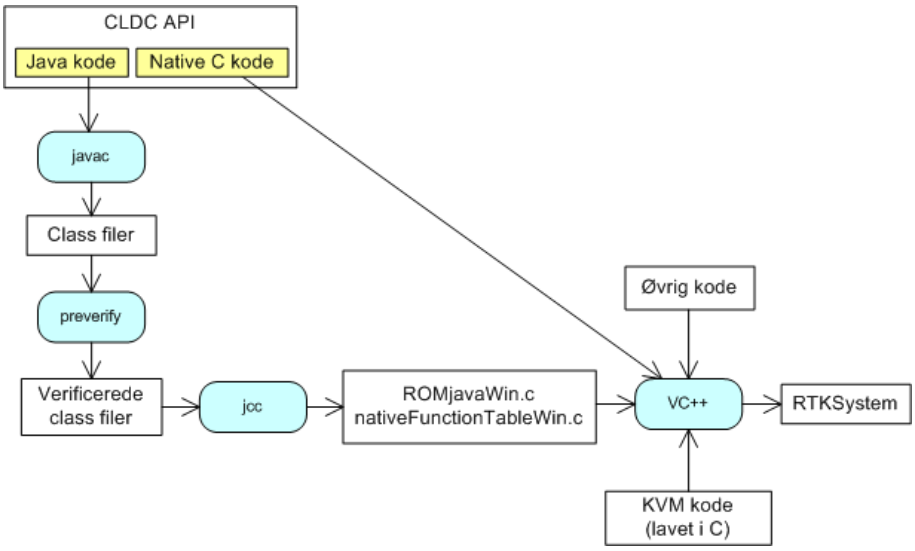
3.1 Bygning af KVM

Et Java-system kan opdeles som vist på figur 3.1. For at Java-programmet kan eksekveres, skal et API være til rådighed.



Figur 3.1: Java system

API'et er implementeret igennem en række native funktioner, som udføres direkte af det underliggende operativsystem. Sun leverer en kompakt og porterbar Java Virtual Machine til Windows-platformen. Denne kan vi bruge direkte på RTKernel.



Figur 3.2: Bygning af KVM

Bygningen af KVM'en følger figur 3.2.

- API'et og eventuelle native-udvidelser kompileres.
- Det kompilerede API preverificeres.
- De preverificerede klasser prelinkes til to C-filer, et ROM-image af API'et og en funktionstabel.

- De to C-filer bruges i et Visual Studio projekt, som kompilerer det endelige system til afvikling på host operativsystemet. Foruden de to filer bruger Visual Studio projektet de C-filer som implementerer KVM'en og eventuelle udvidelser.

For at lette byggeprocessen har vi skrevet en batch-fil. Filen indeholder alle nødvendige kald for at bygge en KVM, og fjerner til sidst midlertidige filer.

```
SET options=RTK_options
SET fileList=RTK_filelist
SET classes_dir=RTK_classes
SET verified_classes=RTK_verified_classes
SET CompactClassPath=tools\jcc\jcc_classes

:clean
RMDIR /Q /S %classes_dir%
RMDIR /Q /S %verified_classes%

del /F /Q nativeFunctionTableRTK.c
del /F /Q ROMjavaRTK.c
del /F /Q RTK_verified_classes.zip

:create
MKDIR %classes_dir%
MKDIR %verified_classes%

:build_api
javac @%options% @%fileList%

:preverify
preverify.exe -classpath %classes_dir% -d
%verified_classes% %classes_dir%

:ZIP
cd %verified_classes%
..\zip -r -q -S ..\%verified_classes%.zip *.*
cd..

:compact
SET classpath=%CompactClassPath%
java JavaCodeCompact -nq -arch KVM -o ROMjavaRTK.c
%verified_classes%.zip
java JavaCodeCompact -nq -arch KVM_Native -o
nativeFunctionTableRTK.c %verified_classes%.zip

del /F /Q %verified_classes%.zip
SET classpath=
```

Figur 3.3: Build.bat

3.2 Programmatisk start af KVM

Før et Java-program kan startes fra et RTKernel-program, skal `UserPath` sættes. Dette angiver stien til Java main-klassen. Hvis klassen ligger i en jar fil angives denne, f.eks: `c:\BTB\Test.jar`. Herefter kan Java programmet startes via kaldet

```
StartKVM(int argc, char* argv[])
```

Den sidste parameter angiver en pointer til et array, der indeholder navnet på den klasse, hvor Java main-metoden findes og parametre til denne. Den første parameter angiver antallet af elementer i arrayet.

3.3 KNI

Et Java-program kan tilgå native funktioner via KNI. Kildekoden til en KNI metode laves i C og eksponeres til Java igennem udvidelser af API'et. C-implementeringen af en KNI funktion afviger fra JNI. Et eksempel på prototypen for en KNI metode i C:

```
KNIEXPORT KNI_RETURNTYPE_BOOLEAN Java_kni_RTKSwitch_on();
```

`KNIEXPORT` angiver, at funktionen kan kaldes fra KVM. `KNI_RETURNTYPE_BOOLEAN` angiver, at funktionen returnerer en Boolean. Den tilsvarende Java funktion ser sådan ud:

```
public native boolean on(int sw);
```

Her ses, at funktionen tager en parameter. Dette fremgik ikke at prototypen for funktionen. KNI lader parametrene ligge på operant-stakken, og de skal „poppes“ direkte derfra. Dette gøres med funktionen `KNI_GetParameterAsInt();`.

Der findes tilsvarende funktioner til at hente andre simple typer fra stakken. Returværdier leveres ligeledes på stakken med funktionen

```
KNI_ReturnBoolean(0);
```

Her returneres false.

4 Test

Vi har løbende testet nye moduler med små testprogrammer fremfor en komplet integrationstest til slut. De efterfølgende afsnit beskriver kort nogle af disse testprogrammer.

4.1 Test af KVM på Windows/RTKernel

Til at teste KVM'en på hhv. Windows og RTKernel brugte vi et simpelt Java program. Dette program skrev en enkelt linie ud på skærmen. Derved fik vi testet at KVM'en kunne starte og indlæse et Java program.

4.2 Test af hardware processer og KNI

Vi testede IO686-kortets hardware via KNI med et Java-program. Det aflæste en DIP-switch og skrev resultatet direkte på den tilsvarende LED samt til `JavaOut`-vinduet. Derved fik vi afprøvet vores native interfaces til hardwaren og `JavaOut`.

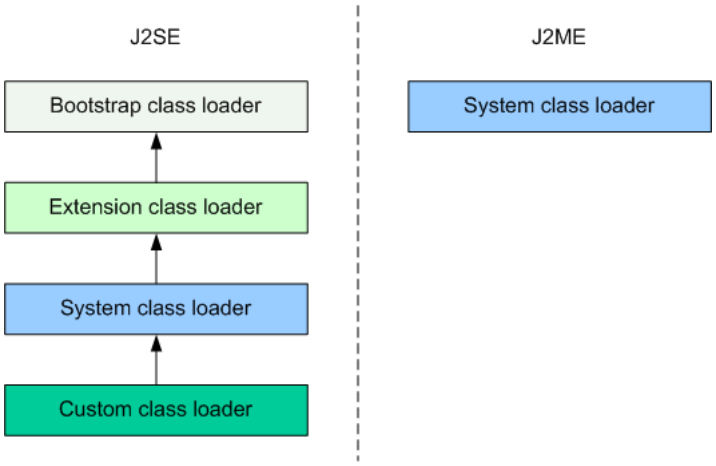
4.3 Test af Java beregningsprogram

Denne test bestod af en gennemkørsel af hovedprogrammet med varierende inputdata på switchen og brug af interruptknappen. Vi kunne observere, at programmet kunne aflæse switchen, foretage en beregning og skrive det korrekte resultat på LED'erne og `JavaOut`-vinduet.

5 Overvejelser

Dette afsnit indeholder de overvejelser, der er gjort i forbindelse med løsning af laboratorieopgave 10.

5.1 Classloader



Figur 5.1: Classloader-arkitektur

5.1.1 Classloaders på J2SE

Class Loader systemet i en J2SE Virtual Machine er bygget op omkring et hierarki af class loaders. Som det fremgår af figur 5.1, er der mulighed for at koble en custom class loader på hierarkiet. Derved kan systemet udvides med nye class loaders (f.eks. til krypterede filer). Der er programmatisk adgang til System class loader og til eventuelle custom class loaders. Derved er man i stand til at indlæse klasser dynamisk. Der kunne evt. udvides med en netværks class loader, der kan loade klasser over netværket.

5.1.2 Classloaders på J2ME

Med et J2ME system har vi meget begrænsede muligheder i forhold til J2SE. Her findes udelukkende en System class loader, som også kunne kaldes en classpath loader. Den er i stand til at loade hovedprogrammet og evt. linkede klasser fra en sti eller jar-fil. Modulet `class.c` sørger for at kalde `loader.c` modulet, når en klasse skal loades fra disken.

Der er ikke programmatisk adgang til System class loader fra Java-programmer. Derved mistes muligheden for at loade klasser dynamisk og for at tilføje custom class loaders. Hvis man alligevel har behov for at udvide class loader funktionaliteten, er der 2 muligheder:

1. Udvide den eksisterende funktionalitet.
2. Genimplementere hele konceptet med inspiration fra metoderne i J2SE.

Er det en simpel ændring med en enkelt ny class loader (f.eks. til indlæsning fra krypterede class-filer med en anden extension), vil det være mest hensigtsmæssigt at udvide den eksisterende class loader. Hvis man har behov for flere nye class loadere og en mere dynamisk brug af dem, vil det være nødvendigt med en større omstrukturering.

5.2 Arbejdsfordeling

Arbejdsfordelingen mellem KVM'en og vores realtids-operativsystem er illustreret i moduloversigten på figur 2.1 på side 3. Det ses på figuren, at den hardware-nære funktionalitet er implementeret i realtidsdelen og eksponeret til KVM'en igennem KNI. Dette er en nødvendighed, da det ikke er muligt at lave low-level IO fra Java-programmel.

På samme måde ville det være nødvendigt at implementere eventuelle tidskritiske funktioner i realtidsdelen, da KVM'en ikke er deterministisk. Ydermere er KVM'en relativt langsom, da denne kun indeholder en bytecode-fortolker men ingen JIT-kompiler.

Selve beregningsdelen af systemet er implementeret i Java. Dette tillader brug af et højniveau programmeringssprog med de fordele dette giver herunder objektorientering, garbage-collection og standardiseret API.

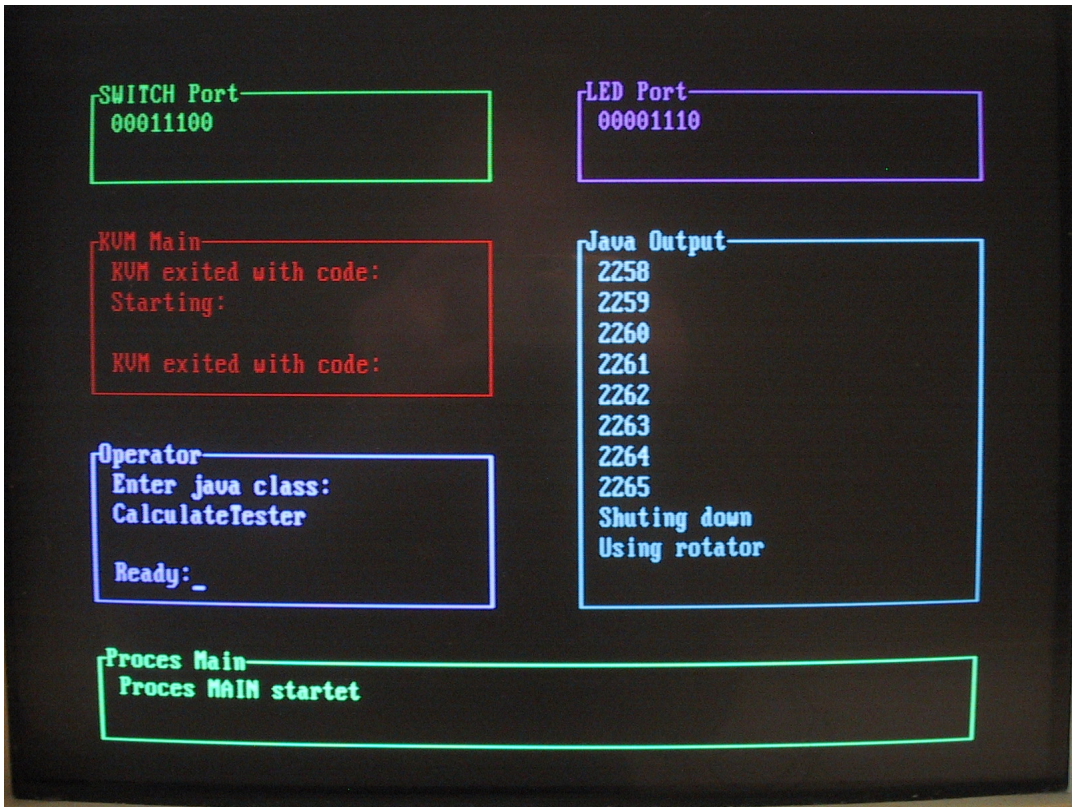
6 Konklusion

Det har været spændende at arbejde med en lille Java implementering. Ved selv at oversætte Sun's reference implementering har vi fået et godt kendskab til sammensætningen af en virtuel maskine. Vi har udbygget Sun's API med funktioner udviklet til et IO686-kort og implementeret native C-kode dertil.

Vi anser kurset som relevant, da der er stor efterspørgel efter ingeniører med erfaring inden for indlejrede systemer. Java gør udviklingen til sådanne systemer lettere og åbner bl.a. op for avanceret netværkskommunikation og grafik.

Arbejdet med projektet har givet en introduktion til KVM og implementeringen bag. Viden om implementeringen er ikke en forudsætning for udviklingen til indlejrede Java-systemer, men det er en stor fordel, da der er mange forskelle mellem J2ME og J2SE.

Bilag



Figur 1: Screenshot