

Examensarbete i Datalogi.

**Korrekt emulering av mikrodator C64 i systemoberoende mjukvara.**

**Correct emulation of micro-computer C64 in software independent of computer system.**

HÅKAN SUNDELL, GU

Handledare, GU: Christer Bernérus  
Examinator, GU: Kent Petersson

GÖTEBORG 1995/96

## Innehåll

Sammanfattning .....	3
Abstract.....	3
Förord .....	4
1. Inledning .....	5
2. Existerande lösningar .....	6
3. Målsättning .....	7
4. Informationsinhämtning .....	8
4.1. Informationsinhämtning angående processor.....	11
4.2. Informationsinhämtning angående timer/in/ut-enhet.....	12
4.3. Informationsinhämtning angående grafikprocessor.....	12
4.4. Resultat från informationsinhämtning angående processor .....	14
4.5. Resultat från informationsinhämtning angående timer/in/ut-enhet.....	14
4.6. Resultat från informationsinhämtning angående grafikprocessor .....	15
5. Emulering .....	17
5.1. Emulering av processor.....	17
5.2. Emulering av busshantering med mera.....	20
5.3. Emulering av grafikprocessor.....	21
5.4. Resultat från emulering .....	21
6. Implementering .....	23
6.1. Implementering för PC.....	23
6.2. Implementering för UNIX/X.....	24
6.3. Resultat från implementering .....	24
7. Testning .....	25
7.1. Resultat från testning .....	25
8. Slutsats.....	26
9. Användardokumentation för C64-emulatorn för PC .....	27
10. Användardokumentation för C64-emulatorn för UNIX/X.....	28
11. Systemdokumentation .....	30
11.1. Systemdokumentation PC .....	34
11.2. Systemdokumentation UNIX/X.....	34
12. Referenser .....	36
13. Ordlista .....	37

## **Sammanfattning**

Avsikten med detta arbete har varit att utröna om korrekt emulering av mikrodatoren C64 i systemoberoende mjukvara är möjlig, och att då tillverka en fungerande implementering.

Jag har utforskat mikrodatoren C64 i detalj så långt det är praktiskt möjligt, och har med stor sannolikhet dokumenterat all nödvändig fakta, både tidigare känd och okänd.

Jag har utarbetat metoder och algoritmer för att skapa ett program som ger korrekt och effektiv emulering i valfri datormiljö.

Resultatet med detta arbete är implementeringar för PC samt UNIX / X, som med få restriktioner har uppvisat mycket god korrekthet. Kärnan i denna implementering kan flyttas till alla datorsystem som använder ANSI C++.

Mitt resultat borde ge en god grund för att utveckla emulatorer av andra mikrodatersystem då dessa oftast har liknande konstruktion som C64.

**Title in english: Correct emulation of micro-computer C64 in software independent of computer system.**

### **Abstract**

The purpose with this work has been to explore if correct emulation of the microcomputer C64 in system independent software is possible, and if so is the case then construct a functional implementation.

I have explored the microcomputer C64 in detail as far as it is practically possible, and i have most certainly documented all necessary information, both previously known and unknown.

I have created methods and algorithms for the making of a program that will give correct and effective emulation in any computer environment.

This work has resulted in implementations for PC and UNIX/X, and these have with a few limitations shown a very good correctness. The kernel in this implementation can be moved to any computer system that uses ANSI C++.

My result should make a good base for developing emulators for other microcomputer systems then these most often has similar construction as C64.

## **Förord**

Detta är ett examensarbete om 20 poäng i Datalogi vid Institutionen för Datavetenskap, Chalmers Tekniska Högskola / Göteborgs Universitet.

Arbetet har utförts under 1995 med cirka 800 arbetstimmar.

Handledare har varit Christer Bernéus.

Examinator har varit Kent Petersson.

## **1. Inledning**

Commodore 64 (C64) är en mikrodator som var mycket populär under åren 1982 - ca 1987. Den har tillverkats i över 10 miljoner exemplar, som är sålda främst i västvärlden. Under senare tid sker försäljningen till en annan marknad, då speciellt till östra europa. Datorerna användes främst till datorspel men det kunde även förekomma begränsat adb-arbete. Det existerar ett mycket stort antal kommersiella program till C64, mer än 10000, och en viss nyproduktion sker fortfarande.

Då dagens datorer för personligt bruk är mycket kraftfullare, är det få personer som fortfarande innehar eller använder C64 i västvärlden. Det finns dock fortfarande ett stort intresse att kunna använda programvaran som finns för C64. Därför finns ett flertal så kallade emulatorer på marknaden, både kommersiella och fria. Det är program som är skrivna för ett visst datorsystem, som möjliggör användning av programvara skrivna för ett annat datorsystem. Dessa har dock gemensamt att endast en bråkdel av all programvara för C64 går att använda, d.v.s de fungerar inte korrekt.

En orsak till detta är att de är interpreterande i realtid och försöker synkronisera exekveringen av C64-programmet med det verkliga tidsflödet. Olika delar av C64-programmet tar olika lång tid att interpretera vilket stör synkroniseringen. Om man skall kunna lyckas med denna emulatorprincip måste man använda ett extremt kraftfullt datorsystem där det för den C64-programdel som interpreteras långsammast tar lika lång tid som om man hade exekverat den på en riktig C64.

En annan orsak till svårigheterna med en korrekt emulering är den bristande mängden information som finns tillgänglig, samt en för liten insats på forskning.

## **2. Existerande lösningar**

Det finns system som emulerar C64 för datormiljöer som PC, Amiga och Unix/X. De hittills bästa systemen finns för PC.

### **C64S:**

- Detta system är skrivet helt i assembler och körs under operativsystemet MS-DOS. Det är helt knutet till den befintliga hårdvaran i en PC. För att nå samma hastighet som en riktig C64 krävs en Intel 486 / 66 Mhz processor eller bättre. Hastigheten för uppdatering av informationen på skärmen är valfri och sker i bestämda tidsintervaller. Avancerade program som kräver total synkronisering mellan processorn och de andra kretsarna ger dels förvrängd grafisk information på skärmen, samt programmet exekveras felaktigt och kan ofta medföra så kallad låsning, även av själva emulatorn. Systemet är kommersiellt och kostar ca 500 kr.

### **PC64:**

- Detta system liknar i stort C64S men kan uppvisa lite bättre korrekthet. Ökningen av korrekthet har gjorts med hjälp av ett antal parametrar. Dessa justeras för hand så att ett visst program exekveras mer eller mindre korrekt. Även detta system är kommersiellt. Tillverkaren tillhandahåller en lista med parametrar passande till olika program.

### **X64:**

- Detta system är skrivet helt i C och körs under operativsystemet UNIX / X-Windows. Alla program som kräver någon som helst synkronisering mellan processorn och de andra kretsarna exekveras helt felaktigt. Detta medför att endast ett litet fåtal program är användbara. Systemet är gjort som ett examensarbete 1991 och är helt fritt.

### **A64:**

- Detta system är skrivet helt i Assembler och körs under operativsystemet AmigaDOS. Det är helt knutet till den befintliga hårdvaran i en Amiga. För att nå samma hastighet som en riktig C64 krävs en Motorola 68030 / 50 Mhz processor eller bättre. Systemet uppvisar lite bättre korrekthet med program som kräver synkronisering än X64. Systemet är kommersiellt.

### **3. Målsättning**

Syftet med mitt arbete är att utarbeta en ny emulatorprincip, då speciellt avsedd för emulering av C64. Denna emulatorprincip skall leda till implementeringar som har så god korrekthet att nästan alla program avsedda C64 exekveras korrekt. Det läggs större vikt på att C64-programmet exekveras korrekt än att kommunikationen med omvärlden sker helt korrekt eller synkront. Det vill säga C64-programmet skall exekveras i en virtuell värld där alla händelser uppfattas internt som synkrona, medan de externt mycket väl kan uppfattas som asynkrona.

## 4. Informationsinhämtning

För att kunna utveckla en princip för emuleringen måste jag ha en korrekt vetskap om hur mikrodatorn C64 fungerar.

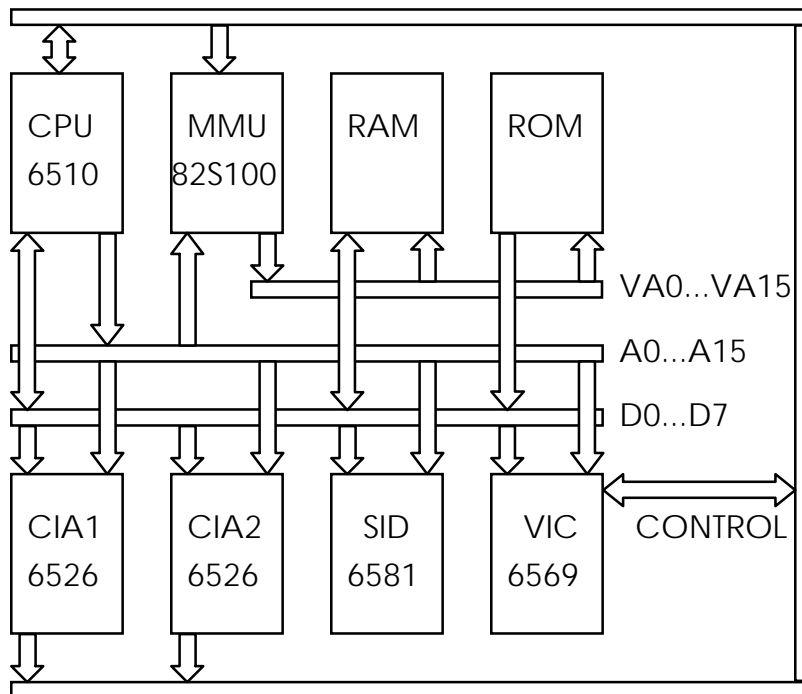
En C64 kan delas in i dessa grundenheter:

- Tangentbord
- Kretskort
- Gränssnitt

Kretskortet består i huvuddrag av dessa enheter:

- Processor (6510)
- Minne (64k ram, 20k rom)
- Timer/In/Ut-Enhet (6526)
- Grafikprocessor (VIC 6569)
- Ljudprocessor (SID 6581)
- Minneshanterare (MMU)
- Adress/Data-buss (16/8 bitar)

Nedan följer ett förenklat blockschema över kretskortet.



Den C64 jag har studerat och tänkt emulera är för PAL systemet. Med detta menas att den är anpassad för det europeiska TV-systemet PAL som bygger på 312 linjer uppdaterade med 50 Hz eller 625 linjer interlaced. Motsvarande system i USA är NTSC som bygger på 262 linjer uppdaterade med 60 Hz eller 525 linjer interlaced.

Adress och databussen är klockad med 985248 Hz (1022727 Hz NTSC) och det är också med denna hastighet processorn och de andra kretsarna arbetar.

Processorn, en MOS6510 är egentligen en MOS6502 med ett inbyggt 8-bitars in/ut-register. Adressbussen är 16-bitars och databussen 8 bitars vilket medför ett adresserbart utrymme på 64 kbyte. Det finns 5 st åtkomstbara 8-bitars register, status SR, stackpekare SP, ackumulator A samt index X och Y. För åtkomst av interna och externa in/ut-register används så kallad memory mapped input/output. Med detta menas att dessa register ses av processorn som vilken minnescell som helst. Dataformatet för 16-bitars värden i minnet är minsta signifikanta byten först. Officiellt finns 56 olika instruktioner och antalet adresseringsmetoder är 13. Det finns två typer av avbrott, ett maskbart (IRQ) och ett icke maskbart (NMI).

För att kunna adressera mer än 64 kbyte finns en minneshanterare. Denna styrs delvis av det interna registret i processorn. Adresseringen kan därför växla mellan bland annat 64 kbyte RAM, 20 kbyte ROM samt 4 kbyte I/O.

Det finns två stycken timer/in/ut-enheter av typen MOS6526. De innehåller vardera två st 8-bitars in/ut-register, två st 16-bitars timers, en klocka, ett skiftregister samt ett maskbart avbrottsregister. Totalt finns 16 st 8-bitars register per krets som kan kommas åt genom bussen. Dessa kretsar kan generera bägge de typer av avbrott som hanteras av processorn.

Grafikprocessorn är av typen MOS6569 och genererar en Video-signal med horisontalfrekvensen 15625 Hz och vertikalfrekvensen 50 Hz (PAL). Upplösningen är konstant, cirka 384 punkter per horisontell linje är synliga på en vanlig TV. Det finns diverse olika valbara text och grafiklägen, 40 gånger 25 tecken med text eller grafik med 320 gånger 200 respektive 160 gånger 200 punkter. Texten eller grafiken kan förskjutas i höjd och sidled. Utöver detta finns även 8 st fritt rörliga objekt som kan överlagras texten eller grafiken, så kallade Sprites. Antalet färger som kan väljas mellan är 16 st, dock med vissa begränsningar vad gäller enskilda bildpunkter. Även denna krets kan generera avbrott, i detta fall maskbart (IRQ). Totalt finns 31 st 8-bitars register som kan kommas åt genom bussen. För att hämta diverse information från minnet om vilka tecken eller grafik som skall ritas ut på skärmen används så kallad Direct Memory Access (DMA). Med detta menas att grafikprocessorn kan tillfälligt hindra processorn från åtkomst till minnet, då den själv använder bussen för åtkomst av minnet.

Ljudprocessorn är av typen MOS6581 och genererar enkanaligt ljud. Internt finns tre stycken ljudgeneratorer som blandas sammas till ett ljud som volymsätts med huvudvolymregistret. För varje ljudgenerator finns en vågformsgenerator som kan generera triangel, sågtand, fyrkant samt brusvågformer i frekvenser från 30 Hz till 12 kHz. Dessutom finns för varje ljudgenerator en programmerbar amplitudgenerator, en så kallad Envelope, med vars hjälp man få olika klangtider för att härma riktiga instrument. Dessa ljud från de tre ljudgeneratorerna kan valfritt blandas samman på olika sätt. Ett programmerbart filter kan även appliceras på det blandade ljudet. Förutom ljudfunktioner finns även två st 8-bitars analog till digital omvandlare. Totalt finns 25 st 8-bitars register som kan kommas åt genom bussen.

Mycket information om processorn och de andra kretsarna finns dokumenterad i böcker som exempelvis Commodore 64 Programmer's Reference Guide. Det finns dock diverse nödvändig information som hittills inte funnits dokumenterad. Därför har jag tvingats att forska fram denna information på egen hand. Då jag inte har haft tillgång till sofistikerad elektronisk mätapparat såsom logikanalysator, har jag istället prövat med att skriva speciella assemblerprogram och har med dess körningsresultat kunnat dra diverse slutsatser.

De kretsar och som jag forskat närmare på är processorn, timer/in/ut-enheterna samt grafikprocessorn. Förutom helt egen forskning har jag verifierat dokumenterad information samt forskat fram ny information då den dokumenterade ibland varit felaktig.

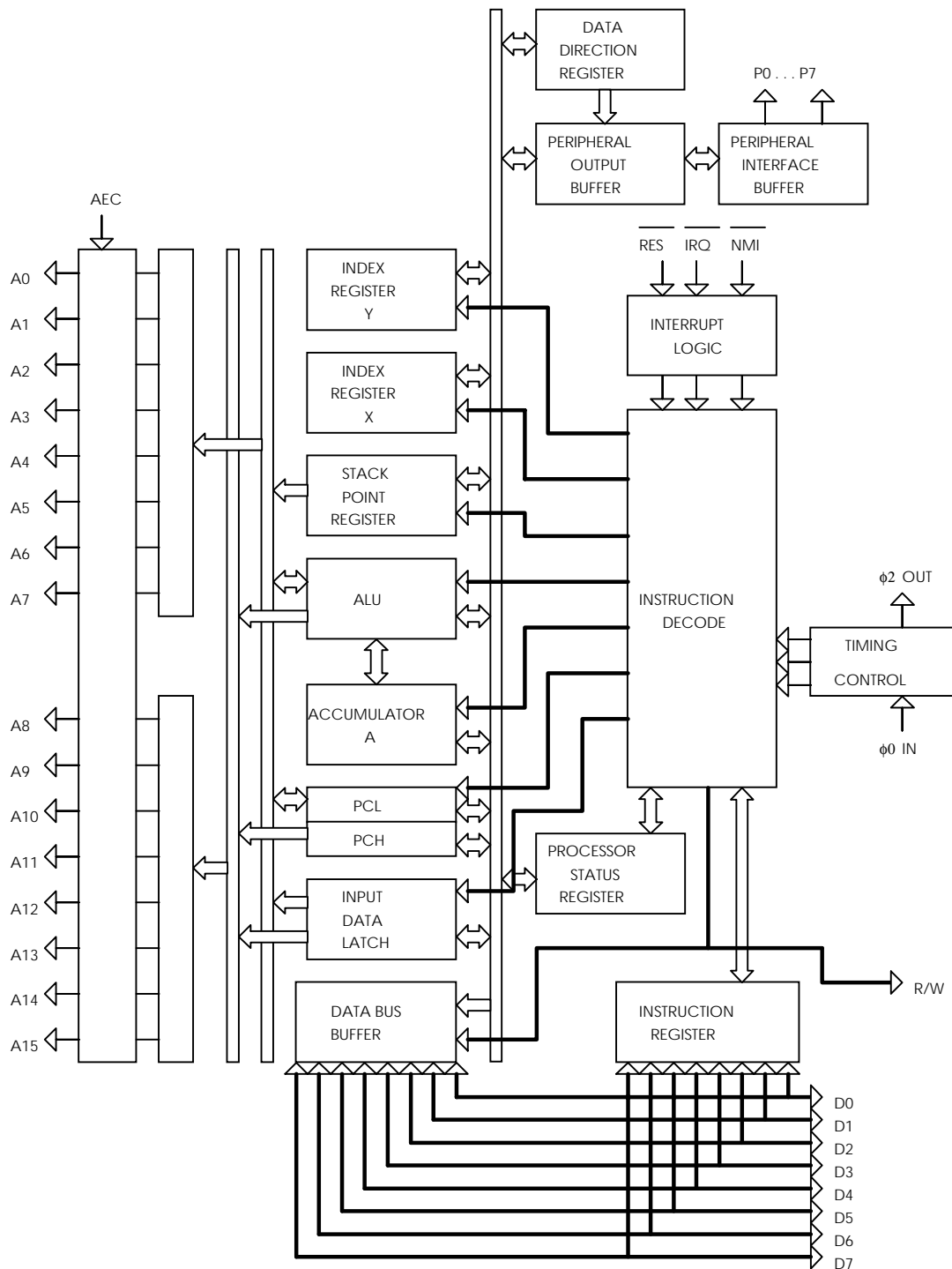
Det som varit oklart angående processorn är avbrottshanteringen, busshanteringen samt hur odefinierade BCD-värden och instruktioner hanteras. Angående avbrottshanteringen ville jag veta exakt hur många maskincykler en avbrottssekvens tar, hur prioriteringar mellan maskbart och icke maskbart avbrotts sköts samt exakt när och på vilka kriterier som en avbrottssekvens utförs. Med busshanteringen menas hur och vad processorn adresserar under en exekvering av en viss instruktion, egentligen studeras hur den så kallade nano-koden i processorn arbetar. I och med att ett 8-bitars BCD-värde endast är definierade för cirka hälften av de värden ett 8-bitars register kan anta, vill jag undersöka hur aritmetiska beräkningar med odefinierade BCD-värden utförs. Ett instruktionsord till processorn är på 8-bitar, men endast 56 av dessa värden är definierade som instruktioner. Därför vill jag undersöka vad som utförs av processorn om man försöker använda något odefinierat värde som instruktionsord.

Timer/in/ut-enheten har som bekant två 8-bitars in/ut-portar. Jag vill undersöka hur dessa tolkas internt när till exempel en hög nivå kopplas till en port som är satt som ingång. Vidare vill jag undersöka vad som händer när man byter räkningsmetod för en timer.

Grafikprocessorn arbetar helt synkront mot processorn. Jag vill veta exakt hur många maskincykler som motsvarar en horisontell linje respektive en hel utritning av skärmbilden. I och med att processorn kan tvinga grafikprocessorn att ändra text eller grafikläge när som helst under uppritningen av skärmbilden kan diverse effekter uppstå. Därför vill jag undersöka vad som händer när man ändrar något av grafikprocessorns register och även exakt när förändringarna syns på skärmen. Vidare vill jag studera hur grafikprocessorn sköter DMA åtkomsten, exakt när denna äger rum och under vilka kriterier.

## 4.1. Informationsinhämtning angående processor

Nedan följer ett blockschema över mikroprocessor MOS 6510.



För att undersöka hur odefinierade BCD-värden hanteras vid aritmetiska operationer som addition och subtraktion har jag använt mig av en så kallad maskinkodsmonitor på C64. Med hjälp av detta program har jag kunnat exekvera sekvenser av assemblerkod och sedan kunnat kontrollera status av register, flaggor och minne före och efter exekveringen. Programmet för kontroll av addition med BCD-värden ser ut som följer.

SED ; Välj aritmetisk läge till att hantera BCD.  
CLC eller SEC ; Nollställer eller sätter minnesflaggan C (Carry).  
LDA #värde1 ; Sätter ackumulatorm till värde1  
ADC #värde2 ; Adderar värde2 till ackumulatorm

Svaret av operation ges i ackumulatorm, dessutom sätts flaggorna Z (Zero), N (Negativ), V (Overflow) samt C (Carry) beroende på resultatet. För att kunna dra någon slutsats av hur hanteringen av odefinierade BCD-värden hanteras har jag testat med olika värden på värde1 respektive värde2 och kontrollerat resultatet. Dessa testresultat har jag sedan infört i en tabell, varefter jag har analyserat denna och kommit fram till en trolig algoritm. För att testa motsvarande för subtraktion har jag bytt ut ADC mot SBC i programmet.

Analysen av odefinierade instruktioner har gått till på liknande sätt. Här har jag dock inte kunnat skriva i assembler utan varit tvungen att skriva in programmet som maskinkod i hexadecimala värden.

För att testa avbrotts hanteringen har jag använt mig av olika program som tvingar fram maskbart respektive icke maskbart avbrott vid olika tider, för att kunna analysera hur prioriteringen mellan dessa avbrott sköts. Den informationsgrund jag arbetat ifrån är från dokumentet angående 6510-processorn av Marko Mäkelä med flera. Dock har detta dokument flera felaktigheter som bland annat rörande avbrotts hanteringen varför jag även varit tvungen att verifiera all information i detta dokument för att inte bygga min forskning på felaktiga grunder.

## 4.2. Informationsinhämtning angående timer/in/ut-enhet

Också här har jag använt mig av en maskinkodsmonitor och diverse små assemblerprogram. Ett av programmen ser ut som följer nedan. Det testar ändring av arbetsläget för Timer A.

```
LDA #värde1
STA $DC04 ; Sätt minsta signifikanta byten av startvärdet för Timer A.
LDA #värde2
STA $DC05 ; Sätt högsta signifikanta byten av startvärdet för Timer A.
LDA #värde3
STA $DC0E ; Sätt arbetsläge för Timer A till värde3
<tidsfördröjning (tom loop)>
LDA $DC04 ; Läs det momentana värdet av minsta signifikanta byten av Timer A.
STA spara1 ; Spara detta värde i minnesadress spara1.
LDA #värde4
STA $DC0E ; Ändra arbetsläge för Timer A till värde4
LDA $DC04 ; Läs det momentana värdet av minsta signifikanta byten av Timer A
STA spara2 ; Spara detta värde i minnesadress spara2.
```

För att testa hur in/ut-portar tolkas internt samt externt har jag satt in/ut-registren till diverse värden, satt konfigurationen för portarna till olika kombinationer av ingångar respektive utgångar. Därefter har jag externt påtvingat respektive läst av olika nivåer.

## 4.3. Informationsinhämtning angående grafikprocessor

Även här har jag använt mig av en maskinkodsmonitor och diverse små assemblerprogram. För att kunna testa vad som händer med utritningen av skärmbilden när man ändrar något av grafikprocessorns register, har jag varit tvungen att utveckla ett program som är helt synkroniserat med utritningen på skärmen. Detta program ser mycket förenklat ut som följer.

```
LDA $D012      ; Läs vilken rad som för närvarande ritas ut på skärmen
CMP värde-1    ; Är detta lika med värde-1
BNE -2         ; Om nej så läs rad igen osv.
LDA värde
STA $D012      ; Sätt registret som jämför och ger avbrott när värde är lika med raden.
NOP           ; Gör ingenting i två maskincykler.
NOP ...        ; Och så vidare...
- efter ett tag kommer ett avbrott genereras och programflödet ändras till
LDX #09
DEX
BNE -2         ; Exekverar en tom loop 9 gånger.
NOP
NOP
LDA $D012      ; Läs vilken rad som för närvarande ritas ut.
CMP värde+1    ; Är detta lika med värde+1
BNE 0         ; Om inte så gör ingenting i en maskincykel.
- här befinner vi oss exakt vid Y=värde+1 samt X=5
```

Därefter exekveras det program som sätter respektive register till olika värden som skall undersökas. Det register som omfattat ytterst mycket undersökningar och analyser är det register som styr DMA åtkomsten av minnet för grafik och textinformation. Jag har för detta register lyckats lista ut den algoritm som grafikprocessorn använder ifråga om detta register. Innan mitt arbete har detta registers påverkan på DMA åtkomsten varit att närmast betrakta som slump.

#### 4.4. Resultat från informationsinhämtning angående processor

Externa avbrott exekveras som nästa instruktion om avbrottet signaleras till processorn och är aktivt senast den näst sista maskincykeln i exekveringen av en instruktion. Alla avbrott tar 7 maskincykler i anspråk. Avbrotten exekveras som följer.

BRK (instruktion, tvingat avbrott):

#	adress	R/W	beskrivning
1	PC	R	läser kod för operationen, öka PC med ett.
2	PC	R	läser och kastar bort, öka PC med ett.
3	0x0100+S	W	skriver bitar 8 till 15 av PC, därefter minska S med ett.
4	0x0100+S	W	skriver bitar 0 till 7 av PC, därefter minska S med ett.
5	0x0100+S	W	skriver statusregistret med B satt till ett, sätt I till ett, därefter minska S med ett, om NMI aktivt så exekveras nästa steg som NMI annars som IRQ.

IRQ samt NMI:

#	adress	R/W	beskrivning
1	PC	R	läser och kastar bort
2	PC	R	läser och kastar bort
3	0x0100+S	W	skriver bitar 8 till 15 av PC, därefter minska S med ett.
4	0x0100+S	W	skriver bitar 0 till 7 av PC, därefter minska S med ett.
5	0x0100+S	W	skriver statusregistret med B satt till noll, sätt I till ett, därefter minska S med ett, om NMI aktivt så exekveras nästa steg som NMI.

IRQ samt BRK:

#	adress	R/W	beskrivning
6	0xFFFFE	R	läser bitar 0 till 7 av PC.
7	0xFFFF	R	läser bitar 8 till 15 av PC.

NMI:

#	adress	R/W	beskrivning
6	0xFFFFA	R	läser bitar 0 till 7 av PC
7	0xFFFFB	R	läser bitar 8 till 15 av PC

Icke maskbart avbrott (NMI) tolkas som aktiv enbart då NMI har signalerats på nytt. Normalt avbrott (IRQ) tolkas dock alltid som aktiv då IRQ signaleras och I är satt till noll.

För hanteringen av BCD-värden har algoritmer utarbetats för både addition och subtraktion. Alla odefinierade instruktioner har gett algoritmer som är verifierade.

#### 4.5. Resultat från informationsinhämtning angående timer/in/ut-enhet

Ändring av arbetsläget för någon timer, det vill säga skrivning till register 14 eller 15, sker enligt följande.

#	R/W	data	beskrivning
0	W	xxx0xxx0	skrivning till 6256
1	-	-	register = data
2	-	-	stanna timer

#	R/W	data	beskrivning
0	W	xxx0xxx1	skrivning till 6526
1	-	-	register = data
2	-	-	starta timer, minska timer med ett

#	R/W	data	beskrivning
0	W	xxx1xxx0	skrivning till 6526
1	-	-	register = data
2	-	-	timer = latch, stanna timer

#	R/W	data	beskrivning
0	W	xxx1xxx1	skrivning till 6526
1	-	-	register = data
2	-	-	timer = latch
3	-	-	starta timer, minska timer med ett

In/ut-portarna tolkas likadant internt (läsning av register) som externt enligt följande.

värde = (utgående | ~styrning) & ingående

#### 4.6. Resultat från informationsinhämtning angående grafikprocessor

Tiden för utritning av en hel skärmsida (312 linjer - PAL) motsvaras av 19656 maskincykler.

Alla register angående färger ger ändring på skärmen omedelbart. Register 22, 24 och bitarna 4 till 6 av register 17 ger ändring med eftersläpning på en maskincykel.

Hanteringen av DMA-åtkomsten fungerar enligt följande algoritm.

```
om (register17[bit 0-2] = linjeräknare[bit 0-2] )
    grafik := 1
```

```
om (register17[bit 0-2] = linjeräknare[bit 0-2] och 12 < kolumnräknare < 45)
    dma := 1
annars
    dma := 0
```

```
om(kolumnräknare = 14 och dma = 1)
    radräknare := 0
```

```
om (register17[bit 0-2] = linjeräknare[bit 0-2] och 15 < kolumnräknare < 45)
{
    teckenbuffer[kolumnräknare-15] := minne[teckenadress]
    färgbuffer[kolumnräknare-15] := minne[färgadress]
}
```

```
om(grafik = 1 och 15 < kolumnräknare < 45)
{
```

```

    teckenadress := teckenadress + 1
    färgadress := färgadress + 1
}

om(kolumnräknare = 45 och grafik = 1)
{
    om(radräknare < 7)
    {
        radräknare := radräknare + 1
        teckenadress := teckenadress - 40
        färgadress := färgadress - 40
    }
    annars
        grafik := 0
}

```

linjeräknare	- Den linje som just nu är under uppritning på skärmen.
kolumnräknare	- Den kolumn som just nu är under uppritning. Det går 63 kolumner på en linje. Kolumnerna motsvarar tidmässigt exakt till maskincyklerna.
dma	- Om aktiv så ägs bussen av grafikprocessorn.
grafik	- Om aktiv så sker utritning av grafisk information denna kolumn.
radräknare	- Den rad (mätt i punkter) av tecknet som är under uppritning.
teckenbuffer	- Denna buffer innehåller information om vilka tecken som skall ritas ut.
färgbuffer	- Denna buffer innehåller information om vilken färg som skall ritas ut.
teckenadress läsas.	- Index till den minnesadress därifrån information om tecken skall läsas.
färgadress	- Index till den minnesadress därifrån information om färger skall läsas.

I de första tre maskincyklerna med DMA aktiv sker ingen egentlig läsning från bussen, i och med att grafikprocessorn måste vänta på att huvudprocessorn avslutar sin åtkomst till bussen. Om grafikprocessorn försöker läsa från minnet under dessa första tre maskinckler blir värdet odefinierat, dock är det oftast hexadecimalt FF.

Med rätt programmering av DMA-hantering kan man åstadkomma effekter som flyttning av informationen i grafikarean i både x och y-led, samt ökad grafisk kvalitet med fler färger per areaenhet.

Vidare har resultat uppnåtts inom följande områden.

- Hanteringen av skärmbblankningen. Om man ändrar kriterierna för denna under rätta tidpunkter kan den enfärgade ramen runt grafikarean på skärmen stängas av, och då tillåta grafisk information att synas utanför den ursprungliga grafikarean.
- Hanteringen av rörliga objekt. Med rätt programmering av dessa rörliga objekt kan man åstadkomma fler än åtta stycken synliga rörliga objekt, samt extra ökning av höjden på objekten.

## **5. Emulering**

Det första som måste bestämmas är vad som skall emuleras. Extern kringutrustning har jag undantagit nästan helt med undantag för TV-skärm, tangentbord och joystick. Det är dock viktigt att emulatorn får en såpass öppen arkitektur att tillägg av stöd för diverse externa enheter är möjligt utan stora förändringar. Vidare undantas emulering av ljudprocessorn tills vidare då det är svårt att finna en standard för ljud som stöds av alla moderna datorsystem.

Grundtanken med emulatorn är att den skall bestå av en kärna, ett objekt i C++. Detta objekt skall vara helt oberoende av datorarkitektur. All kommunikation mellan omvärlden, det vill säga skärmen, tangentbordet och joystick, sker enligt fasta protokoll mellan kärnan och huvudprogrammet. Huvudprogrammet är alltså ett anpassningsprogram som sköter de problem som är specifika för den datorarkitektur som emulatorn skall användas för. Dock har kärnan en intern representation av skärmen, tangentbordet med mera.

De delar som kärnan skall emulera är i stora drag enligt följande.

- Processor.
- Grafikprocessor inklusive generering av skärmbild.
- Minne med minneshanterare.
- Timer/In/Ut-enheter med delar av gränssnitt.
- Bussen med DMA med mera.

De datorsystem som emulatorn är tänkt att fungera för är främst system med en processor, varför parallell programmering borde vara omöjlig eller i annat fall mycket ineffektiv. I och med att jag inte har valt parallell utan instället sekvensiell programmering, måste jag dela upp de parallella förlopp som faktiskt försiggår i C64 sekvensiellt. Det helt dominerande arbetet i datorn utförs av processorn och därför får denna en dominerande ställning också i emulatorns förlopp. De andra enheterna emuleras så att endast akut arbete utförs direkt medan mindre akut arbete läggs på hög och utförs senare i större arbetsblock.

För timer/in/ut-enheten är allt arbete av akut karaktär, och därför måste denna emuleras så att detta hanteras omedelbart när det behövs. Detsamma gäller för minneshanteraren och minnet. Grafikprocessorns arbete består av både akuta och icke akuta delar, varför emuleringen får delas upp i två delar, en för akut samt en för icke akut arbete.

Konstruktionen av emulatorn börjar med processorn, vars konstruktion och principer sätter riktlinjer för hur resterande delar av emulatorn skall konstrueras.

### **5.1. Emulering av processor**

Jag har gjort emuleringen av processorn som ett objekt i C++. Detta objekt innefattar även en del av minnet och minneshanteraren. Den första svårigheten är att resonera fram hur processorns register skall emuleras. De register som finns tillgängliga är som följer.

PC - Programräknare, ett 16 bitars register. Detta register bestämmer från vilken adress i minnet som instruktionsorden skall hämtas.

A - Ackumulator, ett 8 bitars register. Det är med detta register som nästan alla aritmetiska operationer arbetar.

X - Index, ett 8 bitars register. Detta register används för att indexera adresser.

Y - Index, ett 8 bitars register. Detta register används för att indexera adresser.

SP - Stackpekare, ett 8 bitars register. Detta register används för att hålla reda på vilken adress som värden skall sparas eller hämtas på stacken.

SR - Statusregister, ett 8 bitars register. Detta register används för att hålla reda på hur de senaste aritmetiska operationerna har fortlöpt. De saker som registras är om resultatet är lika med noll (Zero), om det är negativt (Negativ), om det blev för stort (Carry) eller om det blev för stort så att det inte går att avgöra om det är negativt eller inte (overflow). Dessutom registreras om tal skall hanteras som BCD-värden vid aritmetiska operationer (Decimal), om maskbart avbrott tillåts (Interrupt) samt om vi har exekverat en BRK instruktion (Break).

Registren PC,A,X,Y och SP kan helt klart emuleras som variabler i C++. Vad gäller SR så är det mycket ineffektivt att konstruera SR efter varje aritmetisk operation då varje flagga (Z,N,C,V,D,I,B) representeras som en bit i SR. Dessutom är det mycket sällan som alla flaggor påverkas av en operation. Därför emuleras varje flagga som en egen variabel i C++, dock med undantag av Z och N som istället emuleras med en gemensam variabel. Detta är effektivast då Z och N sätts samtidigt. Då dessa flaggor alltid bygger på ett 8-bitars resultat, sparas detta resultat i ZN variabeln istället för Z och N flaggorna som kan härledas vid behov. Ännu ett skäl till att emulera varje flagga som en enskild variabel är att villkorliga hoppinstruktioner alltid bygger på tillståndet för en flagga, så gör även många aritmetiska operationer. Hela SR kan dock efterkonstrueras vid behov.

För att emulera processorns exekvering av instruktioner är det mycket lämpligt med switch-case satsen i C++. Det passar mycket bra då varje instruktion består av en byte med noll till två bytes som operander. Detta ger en switch-sats med 256 olika case-satser.

Instruktionsexekveringen skall fortgå tills ett avbrott inträffar, varför en while-sats med stor fördel kan användas som enligt följande skiss.

```
while(!avbrott)
{
    switch(...)
    {
        case 0x00:
            ...
            break;
        ...
        case 0xff:
            ...
            break;
    }
}
```

När avbrott inträffat så avbryts while-slingan och avbrottet hanteras varefter while-slingan återupptas. För den emulerade processorn finns det dock fler undantag än avbrott som skall hanteras. Ett undantag är till exempel att vi vill avbryta exekveringen av instruktioner. Därför modifieras skissen enligt följande.

```
while(!undantag)
{
```

```

...
}
if(avbrott) ...
elseif(stop) ...
...

```

Emuleringen av själva instruktionerna sker mycket strikt i förhållande till hur processorn exekverar dessa i verkligheten. Med detta menas att instruktionerna emuleras i liknande steg som de som utförs internt i processorn. Då många exekveringssteg är mycket lika är det lämpligt att införa subrutiner eller ännu bättre så kallade makro. Med makro menas förkortningar för större programsatser. Här följer ett exempel på emulering av instruktion A9.

```

case 0xa9: /* LDA Immediate - Det vill säga läs den byte som följer efter
instruktionsordet och placera detta värde i ackumulatorm, sedan sätts Z och N
flaggorna beroende på detta värde. */
a = LÄSFRÅNMINNE(pc++); /* a tilldelas värdet från minnet i adress pc, därefter ökas
pc med ett. */
zn = a; /* flaggor Z och N tilldelas värdet i a */
break;

```

Nästa svårighet består i att resonera ut hur emuleringen av minneshantering skall gå till. Ett bra sätt är att typindela minnet. Som sagt tidigare finns det minst tre typer av minne, RAM, ROM och I/O. För varje minnesadress kan man ha ett motsvarande värde i en tabell som beskriver minnestypen. Minneshantering kan dock ändras efter hand programmet exekveras varför man får ha en tabell över olika tabeller av minnestyp. Dessa värden som representerar minnestypen kan även representera mer information. I vårt fall finns endast cirka 100 tillgängliga I/O-register varför vi kan låta minnestypen även representera vilket register som avses. Detta kan vara effektivt då samma I/O-register kan vara åtkomstbar genom olika minnesadresser. Därför kan en läsning från minnet emuleras enligt följande.

```

reg = MinnesTabell[adress];
if(reg == 0) return Ram[adress];
elseif(reg == 1) return Rom[adress];
else return LäsIO(reg);

```

Det är dock inte enbart processorn som kan läsa och skriva till minnet, utan även andra enheter som till exempel grafikprocessorn kan göra åtkomst till minnet genom DMA. Därför måste vi kontrollera om något sådant skall göras innan vi gör läsning eller skrivning till minnet. I och med att dessa DMA åtkomster sker synkroniserat med processorns exekverande måste vi införa en variabel som räknar antalet maskincykler som fortlöpt sedan starten av processorn. Dessutom är det inte bara DMA som sker synkront med processorn, utan andra enheter kan även ändra värdet i något I/O-register eller generera avbrott till processorn vid en viss tidpunkt. Sedan är det så att DMA även hindrar processorn att göra någon som helst exekvering under ett visst antal maskincykler. Så åtkomstkontrollen ser ut som följer.

```

maskinckyel++; /* Nästa maskinckyel */
if(maskinckyel == händelsecykel)
{
    maskinckyel = KollaHändelse(maskinckyel);
    händelsecykel = NästaHändelse(händelsecykel);
}

```

```
}
```

## 5.2. Emulering av busshantering med mera

Denna del emulerar det som inträffar när processorn vill göra en åtkomst eller åtkomstkontroll till bussen med eventuell hantering av någon händelse. Vidare kan man säga att den emulerar det akuta arbete som skall utföras för någon av de andra enheterna.

Då objektet som emulerar processorn även hanterar en viss del av åtkomsten till minnet, återstår emuleringen av åtkomsten till de olika enheternas I/O-register. Här följer huvuddragen i emuleringen för läsning av I/O-registren.

```
switch(register)
{
  case 1:          /* Exempelvis Grafikprocessorn, bakgrundsfärg */
    return bakgrundsfärg;
  case 2:
    return ...
  ...
}
```

Vid skrivning till diverse I/O-register kan det hända att det uppkommer icke akut arbete som kan utföras senare. Då löses detta genom att använda olika stackar. På dessa stackar sparas information om vilket register som skrevs till, vad som skrevs samt när (vilken maskincykel). Det är lämpligt att ha olika stackar för olika enheter, så att när emuleringen för detta icke akuta arbete för en viss enhet exekveras endast har en stack att hämta information ifrån.

Händelser hanteras på ett liknande sätt, då även dessa kan resultera i icke akut arbete. Då sparas även information om detta arbete på stackar. Ofta är detta icke akuta arbete av karaktären att det inte måste utföras för att processorn skall exekvera programmet korrekt, utan endast för att ge information till omvärlden, till exempel skärmen.

För att få emuleringen av händelser att bli effektiv så delas dessa upp inom olika områden, bland annat efter vilken enhet de härrör ifrån. Emuleringen av händelsehanteringen är enligt följande.

```
switch(grundtyp)
{
  case 1:          /* Exempelvis att någon enhet gör DMA åtkomst */
    <stanna processorn i x antal maskincycler>
    break;
  case 2:          /* Grafikprocessor */
    switch(spectyp)
    {
      case 1:      /* generera maskbart avbrott IRQ */
        ...
        break;
      case 2:
        ...
    }
}
```

```
}  
case 3:  
...  
}
```

De grundtyper av händelser som hanteras är stopp, DMA, undantag, grafikprocessor, inkommande data, timer-enhet 1 samt timer-enhet 2.

Varför det är lämpligt att dela upp händelserna enligt detta är att vi måste sortera händelserna efter vilken maskincykel som händelsen skall inträffa varje gång som någon ny händelse dyker upp. Den händelse som inträffar närmast den maskincykel som processorn exekverar för tillfället är den händelse vars maskincykel skall jämföras varje gång processorn gör en åtkomstkontroll. Generering av nya händelser kan komma från bland annat skrivning eller läsning av I/O-register, eller som en följd av redan hanterade händelser.

### 5.3. Emulering av grafikprocessor

Denna del emulerar grafikprocessorernas generering av visuell grafikinformation. Detta arbete är att betrakta som icke akut samt är dessutom inte nödvändigt för korrekt exekvering av c64-programmet. Den akuta delen av arbetet, det vill säga grafikprocessorernas inhämtning av data emuleras av busshantering.

Genereringen av grafikinformation sker radvis, med 504 punkter i varje rad. För att få genereringen att bli så effektiv som möjligt ritas varje rad i två steg. Först ritas den grafik som ligger som en bakgrund sett ur synpunkt från de rörliga grafiska objekten. Sedan ritas den grafik som härrör till de rörliga grafiska objekten över den ursprungliga bakgrundsgrafiken. Ritningen sker enbart i en tabell i minnet över punkterna i raden, den visuella utritningen sköts av det för implementationen specifika programmet.

Informationen för utritningen hämtas från de stackar som genereras av emuleringen av busshantering. Två stackar används, en för bakgrundsgrafiken samt en för de rörliga grafiska objekten. I och med att informationen som är sparade på stackarna är märkt med exakt tidpunkt i förhållande till grafikprocessorn, är det bara att plocka information från stackarna varefter utritningen fortskrider.

### 5.4. Resultat från emulering

Kärnan i emulatorn är ett objekt i C++ helt oberoende av datorsystem. All kommunikation mellan den riktiga och den virtuella värld som representeras i kärnan görs genom väl specificerade funktioner och dataformat.

Starten av emuleringen understöds av funktioner för total återställning och åtkomst av minnet.

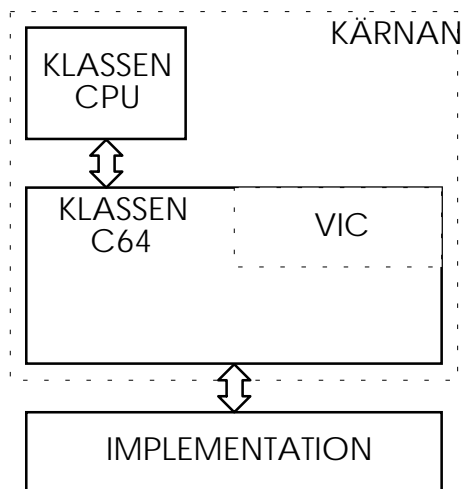
Själva emuleringen understöds av funktioner för exekvering av visst antal maskinnycklar, rapportering av yttre enheter som tangentbord med mera, samt generering av grafisk information.

För tangentbordet och yttre enheter rapporteras endast förändringar, då emulatorn har en egen intern representation av dessa. Tangentbordet representeras som en lista av 8 bytes eller informellt en matris av 8 gånger 8 bitar, där varje bit innehåller information om tangenten är nertryckt (0) eller uppsläppt (1). Bitarna i matrisen har betydelse enligt följande skiss.

	Bit 7	6	5	4	3	2	1	0
Index 7	RUN STOP	Q	C=	SPACE	2	CTRL	ARROW LEFT	1
6	/	ARROW UP	=	RIGHT SHIFT	CLR HOME	;	*	£
5	,	@	:	.	-	L	P	+
4	N	O	K	M	0	J	I	9
3	V	U	H	B	8	G	Y	7
2	X	T	F	C	6	D	R	5
1	LEFT SHIFT	E	S	Z	4	A	W	3
0	CRSR DOWN	F5	F3	F1	F7	CRSR RIGHT	RETURN	INST DEL

Genereringen av grafisk information sker radvis och raden representeras som en lista av heltal, där varje heltal motsvarar färgen på en bildpunkt.

Nedan följer ett blockschema över kärnan i emulatorn och en eventuell implementation.



## **6. Implementering**

Emulatorn har implementerats för PC samt för UNIX/X.

Ett stort problem vid implementeringen var svårigheter att få fram dokumentation över datorsystemen. För PC kunde det även innebära inkorrekt information. Detta löstes genom att söka genom Internet, vilket dock tog mycket tid i anspråk.

För implementeringen behövs ett antal funktioner konstrueras som är specifika för det avsedda systemet. De funktioner som avses är följande.

- Utritning av grafik på bildskärmen, det vill säga en funktion som förmedlar visuellt den grafiska information som emulatorn genererat till omvärlden.
- Avläsning av tangentbordet. Detta är i själva verket en emulator för tangentbordet som emulerar den befintliga inmatningsenheten som ett tangentbord för C64.
- Avläsning av yttre enheter. Denna funktion emulerar joystick med mera genom användning av de befintliga resurserna för systemet.
- Realtidsklocka. För synkroniseringen av emulatorn med omvärlden behövs en klocka som har en noggrannhet av 20 ms.
- Initiering av ROM och RAM. Innan emulatorn kan exekveras måste ett operativsystem och eventuellt ett program laddas in i emulatorns representation av minne.

Huvudprincipen för alla implementationer av emulatorn ser i stort ut som följer.

1. Initiera emulatorkärnan.
2. Läs in operativsystem och program, skicka dessa till emulatorkärnan.
3. Låt emulatorkärnan exekvera ett antal maskin-cykler motsvarande en skärmutritning.
4. Om någon yttre enhet som tangentbord med mera har förändrats, så rapporteras detta till emulatorkärnan.
5. Låt emulatorkärnan generera en rad med grafisk information och rita ut denna på bildskärmen.
6. Om det har gått mindre än 20 ms sedan steg 3 utfördes, så utför steg 5 igen.
7. Om exekveringen av emulatorn skall fortsätta, så utför steg 3 igen.

### **6.1. Implementering för PC**

Implementeringen för PC har skrivits i C++ och assembler. Jag har valt att skriva programmet för MS-DOS då grafikhanteringen i Windows är förhållandevis mycket långsam. En orsak till varför vissa delar har skrivits i assembler är att det finns ett mycket begränsat stöd för grafik, tangentbord med mera i C++ för MS-DOS. Dessutom är det mycket betydande för hastigheten att skriva vissa hårdvarunära delar i assembler.

Jag har valt att använda hela skärmytan för visning av grafiken. Den synliga bildstorleken för en C64 är 384 gånger 282 punkter. Vanlig VGA har upplösningen 320 gånger 200 punkter.

Dock så går hårdvaran för VGA att programmera om så att upplösningen för C64 erhålls. På grund av kompatibilitetsproblem mellan olika VGA-grafikkort och bildskärmar har jag konstruerat 4 olika grafiklägen. De är 320 gånger 200, 368 gånger 240, 384 gånger 246 samt 384 gånger 282 punkter.

Tangentbordet implementeras helt i hårdvara. En tangent på tangentbordet i C64 motsvaras direkt av en tangent på tangentbordet för PC. Även joystick implementeras som tangenter.

Ett problem som uppstod i samband med implementeringen var att MS-DOS endast kan hantera 640 kbyte minne på ett användbart sätt. Dessutom kan inte kompilerade avsnitt med programkod eller listor med data vara större än 64 kbyte. Detta ledde till att kärnan inte kunde kompileras. För att lösa dessa problem fick jag använda mig av en så kallad DOS-extender (DPMI). Detta program möjliggör program som använder en linjär minnesmodell att exekveras under DOS.

## 6.2. Implementering för UNIX/X

Implementeringen för UNIX/X har skrivits i C++. Ingen assembler har behövt skrivas då Unix och X-Windows tillhandahåller tillräckliga funktioner.

För visning av grafiken används ett fönster i X-Windows med storleken 384 gånger 282 eller 768 gånger 564 punkter. Den senare storleken av fönster visar varje bildpunkt för C64 som 2 gånger 2 punkter. Den ökade fönsterstorleken kan behövas då en normal bildskärm för X-Windows ofta har en upplösning på 1024 gånger 768 punkter, och då blir grafiken otydlig i den lilla fönsterstorleken. Då färger skall visas krävs en X-Server med stöd för 256 färger. X-Servern kan vara ganska långsam då grafikinformationen skickas över ett datornät. För att lösa detta har jag konstruerat ett så kallat cache-minne som representerar hela bildskärmen. Endast när ny grafikinformation skiljer sig från cache-minnet så skickas denna till X-Servern.

Tangentbordet implementeras helt i hårdvara. En tangent på tangentbordet i C64 motsvaras direkt av en tangent på tangentbordet för arbetsstationen. Även joystick implementeras som tangenter. Då vissa icke alfanumeriska tangenter inte finns på alla tangentbord får dessa tangenter implementeras olika för olika tangentbord. För detta ändamål finns en textfil som beskriver konfigurationen för tangentbordet, denna textfil får anpassas till det aktuella tangentbordet.

## 6.3. Resultat från implementering

Jag har åstadkommit två stycken implementeringar, en för PC med MS-DOS och en för UNIX med X-Windows. Dessa implementeringar är förhållandevis lika, både i funktionalitet samt i uppbyggnad. Cirka 90 procent av källkoden för dessa implementationer är gemensam och dessa utgörs helt av den så kallade kärnan i emulatorens. Denna kärna är identisk i bägge implementeringarna då endast tillägg till kärnan behövdes.

För UNIX/X har emulatorens kompilerats för två olika datorarkitekturer. Dessa är SUN-arbetsstationer med SPARC-processor, samt för datorarkitekturer byggda på DEC ALPHA-processor.

## **7. Testning**

För att testa emulatoren har jag använt mig av ett mycket stort antal C64-program. Dessa program har jag överfört från det media som C64 använder sig av som standard, kassettband. För att överföra programmen har jag använt specialkonstruerad hårdvara till datorsystemet AMIGA, som möjliggör anslutning av den speciella bandspelare som används till C64.

Programmen är i huvudsak spel, men även program avsedda för programutveckling och avlusning på C64, så kallade maskinkodsmonitorer har testats. Med hjälp av dessa har jag kunnat testa emuleringen mycket ingående på hårdvarunära nivå, då dessa maskinkodsmonitorer kan visa i stort sett all information och status om hårdvaran i C64.

Jag har även använt ett kommersiellt avlusningsprogram för UNIX som heter Purify.

Testningen har lett till att ett flertal fel i emulatoren har blivit åtgärdade med tillägg och ändringar. Testningen har upprepats till dess inga fel har kunnat påvisas.

Ett problem med testningen var att vissa fel som upptäcktes, oftare berodde på fel i dokumentationen över C64 än rena programmeringsfel, vilket ledde till mer forskning med ökad arbetsinsats som följde.

### **7.1. Resultat från testning**

Med hänsyn undantagen till ljud har följande program uppvisat total korrekthet.

Program som inte gör någon åtkomst till andra filer eller externa enheter förutom joystick. Minst 99 procent av alla program till C64 är av denna typ.

Övriga program som kräver externa enheter har uppvisat total korrekthet i jämförelse med en original C64 utan dessa externa enheter.

För att få korrekt hastighet på implementeringen för PC krävs som lägst en processor av typ 486DX2/66. Snabbare processor ger snabbare uppdatering av den grafiska informationen.

För att få korrekt hastighet på implementering för UNIX/X krävs som lägst en processor av typ SPARC 40 Mhz. Snabbare processor ger snabbare uppdatering av den grafiska informationen. Körning på datorer med processor av typ DEC ALPHA ger en mycket god hastighet, beräkningsmässigt lika snabb uppdatering som original C64. Dock så begränsas uppdateringen av grafisk information av nätverk och X-Servern.

## **8. Slutsats**

Commodore C64 har en relativt komplicerad hårdvara. Denna hårdvara är nu dokumenterad i stort sett alla avseenden vad gäller funktionalitet.

Emulering av C64 är möjlig, dessutom är det möjligt att tillverka en kärna av emulaton som är systemoberoende. En sådan kärna har tillverkats, skriven i programspråket ANSI C++. Grundideerna i denna kärna bör vara användbar i konstruktion av emulatorer av andra äldre mikrodatorsystem då dessa ofta har liknande arkitektur.

Två implementeringar av en emulator för C64 har gjorts, en för PC samt en för UNIX/X. Båda är byggda på den tillverkade emulatorkärnan. För implementeringen på PC används MS-DOS.

Implementeringen på UNIX/X har kompilerats på två olika datorarkitekturer, SUN SPARC och DEC ALPHA. Kompilering bör kunna ske på samtliga UNIX/X arkitekturer, med endast små ändringar i Makefile.

Implementeringar på de flesta andra moderna datorsystem bör vara möjlig med en relativt liten arbetsinsats.

Emulaton för C64 har med få restriktioner uppvisat mycket god korrekthet.

## 9. Användardokumentation för C64-emulatorn för PC

Detta avsnitt beskriver hur man använder PC-versionen av C64-emulatorn.

Systemkrav:

- Processor Intel 486 33 Mhz eller snabbare
- 4 Mb RAM
- MS-DOS version 5.0 eller nyare
- VGA

För att starta emulatorn utan ett förinladdat C64-program, skriv C64 på kommandoraden.

För att starta emulatorn med ett förinladdat C64-program, skriv C64 samt därefter filnamn på kommandoraden. De filer med C64-program som understöds har filnamn med suffixen .PRG eller .T64.

När emulatorn är igång fungerar tangentbordet som tangentbordet på en riktig C64 med dessa modifikationer.

Pause	- Avsluta emulatorn
Vänstra Ctrl	- C64 Commodore
Escape	- C64 Run/Stop
Delete	- C64 Arrow Up
Insert	- C64 £
Home	- C64 Clr/Home
Högra Alt	- Joystick 1 Fire
Numerisk %	- Joystick 1 Uppåt
Numerisk 5	- Joystick 1 Neråt
Numerisk 7	- Joystick 1 Vänster
Numerisk 9	- Joystick 1 Höger
Högra Ctrl	- Joystick 2 Fire
Numerisk 8	- Joystick 2 Uppåt
Numerisk 2	- Joystick 2 Neråt
Numerisk 4	- Joystick 2 Vänster
Numerisk 6	- Joystick 2 Höger

För att ändra grafikläge, editera filen emulator.cfg och ändra enligt följande:

- 1 - 384 gånger 282 punkter, läge "Chained" i 50 Hz.
- 2 - 384 gånger 246 punkter, läge "Chained" i 60 Hz.
- 3 - 368 gånger 240 punkter, läge "Chained" i 60 Hz.
- 4 - 384 gånger 282 punkter, läge "Planar" i 50 Hz.
- 5 - 384 gånger 246 punkter, läge "Planar" i 60 Hz.
- 6 - 368 gånger 240 punkter, läge "Planar" i 60 Hz.
- 7 - 320 gånger 200 punkter, läge "Chained" i 70 Hz.

## 10. Användardokumentation för C64-emulaton för UNIX/X

Detta avsnitt beskriver hur man använder UNIX/X-versionen av C64-emulaton.

Systemkrav:

- X-Windows system med X-Server terminal kapabel att visa 256 färger.
- Terminal med tangentbord

Om programmet inte är kompilerat, det vill säga avsaknaden av en fil vid namn c64 som är körbar för det avsedda systemet, kör programmet make. Eventuellt kan det krävas ändringar i filen Makefile om sökvägarna inte stämmer.

För att starta emulaton utan ett förinladdat C64-program, skriv c64 på kommandoraden.

För att starta emulaton med ett förinladdat C64-program, skriv c64 samt därefter filnamn på kommandoraden. De filer med C64-program som understöds har filnamn med suffixen .PRG eller .T64.

Fler alternativ som kan fås genom att lägga till detta till kommandoraden är följande.

- 1 Litet fönster (384 gånger 282 punkter)
- 2 Stort fönster (768 gånger 564 punkter)
  
- sharemem Använder sig av en utvidgning från MIT för grafiken. Kan enbart användas om programmet körs på samma terminal som visar grafiken då det kan snabba upp hastigheten på grafikuppdateringen väsentligt.
  
- cache Använder sig av en intern buffert för grafiken. Detta kan snabba upp hastigheten på grafikuppdateringen väsentligt i och med att endast ny grafik sänds till X-Servern. Dessutom används mindre systemresurser som till exempel nätkapacitet.
  
- tilt Vrider fönstret 90 grader medurs. Kan vara användbart vid applikationer för C64 som kräver att TV eller skärm ställs på kanten.

Här följer en komplett syntax för kommandoraden.

```
c64 [-12] [-sharemem] [-cache] [filename(.prg/.t64)]
```

När emulaton är igång samt muspekaren står över fönstret fungerar tangentbordet som tangentbordet på en riktig C64 med avseende på alla normala alfa-numeriska tangenter. För resterande för C64 speciella tangenter används filen keys. Denna fil definierar för varje speciell tangent motsvarande tangentbordskod på den befintliga terminalen. Vid byte till terminal med annan typ av tangentbord kan dessa koder behövas att ändras. Dessa koder är hexadecimala och har följande beteckningar och betydelse.

- keyPlus: - Plus ( + )
- keyMinus: - Minus ( - )
- keyEqual: - Lika med ( = )

keyPound:	- Pund ( £ )
keyUpArrow:	- Pil uppåt ( ↑ )
keyAsterisk:	- Asterisk ( * )
keyAt:	- Alfa ( @ )
keyColon:	- Kolon, Vänsterklammer ( : , [ )
keySemiColon:	- Semikolon, Högerklammer ( ; , ] )
keyDivide:	- Delat med, Frågetecken ( / , ? )
keyLeftArrow:	- Pil vänster ( ← )
keyCommodore:	- Commodore ( C= )
keyRunStop:	- Run, Stop ( RUN/STOP )
keyCtrl:	- Control ( CTRL )
keyRestore:	- Restore ( RESTORE )
keyClrHome:	- Clear, Home ( CLR/HOME )
keyInstDel:	- Insert, Delete ( INST/DEL )

För joystick används det numeriska tangentbordet samt Metatangenterna enligt följande.

Vänstra Meta	- Joystick 1 Fire
Numerisk %	- Joystick 1 Uppåt
Numerisk 5	- Joystick 1 Neråt
Numerisk 7	- Joystick 1 Vänster
Numerisk 9	- Joystick 1 Höger
Högra Meta	- Joystick 2 Fire
Numerisk 8	- Joystick 2 Uppåt
Numerisk 2	- Joystick 2 Neråt
Numerisk 4	- Joystick 2 Vänster
Numerisk 6	- Joystick 2 Höger

## 11. Systemdokumentation

Först beskrivs den systemberoende kärnan.

Kärnan består sett från en synvinkel för implementation av en klass C64. Denna är indelad i tre huvudsakliga block, CPU, EMULATOR och VIC.

Blocket CPU utgörs av en klass CPU och är relativt fristående från resten av kärnan. Dock så måste det ha tillgång till blocket EMULATOR med klassen C64. Dessutom måste klassen C64 ha direkt tillgång till klassen CPU då vissa funktioner i blocket EMULATOR kan ge sidoeffekter i klassen CPU. Definitioner av blocket CPU finns i filen `cpu.h`, samt programdelen finns i filerna `cpu.cpp` och `cpu_defs.h`.

För extern kommunikation har klassen CPU följande funktioner.

`CPU(C64 *c64)` - Konstruerare för klassen CPU, argumentet `c64` är en pekare till huvudklassen C64.

`unsigned int Run(unsigned int cyclesleft)`  
- Exekverar emulering av mikroprocessor 6510 tills klassen C64 rapporterar ett stopp. Argumentet `cyclesleft` beskriver antalet maskincykler som återstår att exekvera tills en händelse skall hanteras. Efter ett rapporterat stopp så returneras antalet maskincykler som återstår tills nästa händelse.

`void Set6510Regs(UBYTE a,UBYTE x,UBYTE y,UBYTE sr,UBYTE sp,UWORD pc)`  
- Tilldelar den interna representationen av mikroprocessorns register enligt respektive argument.

`void Get6510Regs(UBYTE *a,UBYTE *x,UBYTE *y,UBYTE *sr,UBYTE *sp,UWORD *pc)`  
- Tilldelar argumenten värdet av respektive register i representationen av mikroprocessorn.

Funktionen `Run` använder sig av följande interna underfunktioner.

`unsigned int DecimalAdd(unsigned int alu,unsigned int a,unsigned int c)`  
- Utför addition med BCD-talen i argumenten `alu` och `a` samt med carry-flaggan i argumentet `c`. Svaret returneras samt flaggorna sätts i den globala strukturen `decimalregs` av typen `DecimalRegs`.

`unsigned int DecimalSub(unsigned int alu,unsigned int a,unsigned int c)`  
- Utför subtraktion med BCD-talen i argumenten `alu` och `a` samt med carry-flaggan i argumentet `c`. Svaret returneras samt flaggorna sätts i den globala strukturen `decimalregs` av typen `DecimalRegs`.

```
den
struct DecimalRegs
{
    unsigned int zn;
    unsigned int a;
    unsigned int c;
    unsigned int v;
};
```

Blocket EMULATOR utgörs av en klass C64 och är grundstommen i kärnan. Dock så måste det ha tillgång till blocket CPU med klassen CPU. Definitioner av blocket EMULATOR finns i filen emulator.h, samt programdelen finns i filerna emulator.cpp. Observerara att även blocket VIC inbegrips av klassen C64.

För extern kommunikation har klassen C64 följande funktioner.

C64() - Konstruerare för klassen C64. Denna funktion konstruerar även ett objekt av klassen CPU.

void Reset() - Initierar emulatorns alla delar motsvarande en kallstart av C64.

UWORD Run6510(UWORD cykler) - Exekverar emulering av C64 systemet i antalet maskincykler motsvarande argumentet cykler. Funktionen returnerar den maskin cykel, refererat till grafikprocessorn, som exekveringen stannar vid.

void SetKey(const unsigned int key) - Ändrar den interna representationen av tangentbordet. Argumenten key beskriver vilken tangent som har ändrats. Informationen i key tolkas enligt följande. Bit 7 indikerar om tangenten var nertryckt (1) eller uppsläppt (0). Bitarna 6-4 är kolumnen för tangenten. Bitarna 2-0 är motsvarande rad för tangenten. Bit 3 indikerar om tangenten skall tolkas som om den var kombinerad med shift (1) eller inte (0).

void SetJoystick(const UBYTE joy, UBYTE dir) - Ändrar den interna representationen av joystick nummer motsvarande argumentet joy. Riktningen på joysticken sätts enligt argumentet dir. Informationen i dir tolkas enligt följande. Bit 4 indikerar om fireknappen är nertryckt (1) eller inte (0). Riktningsslägena indikeras motsvarande i bit 3 för höger, bit 2 för vänster, bit 1 för neråt samt bit 0 för uppåt.

void SetPaddles(const UBYTE nr, const UBYTE x, const UBYTE y) - Ändrar den interna representationen av paddles nummer motsvarande argumentet nr. Läget på paddlen sätts enligt argumenten x och y.

void SetRom(const UWORD addr, const UWORD len, UBYTE\* buffer) - Kopierar till den interna representationen av ROM-minnet antalet motsvarande argumentet len från och med adressen enligt argumentet addr. Källan till kopieringen är en lista av bytes enligt argumentet buffer.

void SetRam(const UWORD addr, const UWORD len, UBYTE\* buffer) - Kopierar till den interna representationen av RAM-minnet antalet motsvarande argumentet len från och med adressen enligt argumentet addr. Källan till kopieringen är en lista av bytes enligt argumentet buffer.

Funktionen Reset använder sig av följande interna underfunktioner.

void InitMMU() - Konstruerar den interna representationen av minneshanteraren. Den består av åtta listor av bytes. Varje position i listan indikerar minnestypen för motsvarande adress i minnet.

Följande funktioner är interna underfunktioner som anropas av klassen CPU.

unsigned int ReadIO(const unsigned int reg)

- Läser en byte från en I/O-enhet från register enligt argumentet reg. Den kompletta adressen är enligt variabeln addr i klassen CPU. Svaret returneras samt antalet maskinckler som återstår tills nästa händelse sätts i variabeln cyleft i klassen CPU.

unsigned int WriteIO(const unsigned int data)

- Skriver en byte enligt argumentet data till en I/O enhet till register enligt variabeln reg i klassen CPU. Den kompletta adressen är enligt variabeln addr i klassen CPU. Funktionen returnerar antalet maskinckler som återstår tills nästa händelse. Om skrivningen får en effekt som inte kan genomföras direkt, sparas information om skrivningen på stackarna gfxsavedata eller spriteSaveData som är globala listor av typ GfxEvent respektive SpriteEvent. Vilken stack som används beror på om effekten rör de rörliga grafiska objekten (SpriteEvent) eller inte (GfxEvent).

```
struct GfxEvent
```

```
{  
    unsigned int cycle;  
    unsigned int type;  
    unsigned int data;  
};
```

```
struct SpriteEvent
```

```
{  
    unsigned int cycle;  
    unsigned int next;  
    unsigned int type;  
    ULONG data;  
};
```

unsigned int WannaRead()

- Kontrollerar om bussen är ledig för läsning. Om den inte är ledig eller om en händelse skall hanteras görs detta tills bussen är ledig för

läsning. Funktionen returnerar antalet maskinckler som återstår tills nästa händelse.

unsigned int WannaWrite()

- Kontrollerar om bussen är ledig för skrivning. Om den inte är ledig eller om en händelse skall hanteras görs detta tills bussen är ledig för skrivning. Funktionen returnerar antalet maskinckler som återstår tills nästa händelse.

void SetExceptionNextCycle(unsigned int)

- Sätter att avbrott skall rapporteras som en händelse nästa maskinckel. Argumentet är antalet maskinckler som återstår tills nästa händelse.

unsigned int SortEvents(unsigned int)

- Sorterar händelser efter tidpunkt då de skall rapporteras. Tar som argument och returnerar antalet maskinckler som återstår tills nästa händelse.

Följande funktioner är interna underfunktioner som anropas av funktionerna WannaRead och WannaWrite.

void CheckEvent() - Hanterar händelser

void SortEvents(void) - Sorterar händelser efter tidpunkt då de skall rapporteras.

void SortSpriteDma(const unsigned int)  
 - Sorterar de rörliga grafiska objekten efter tidpunkt då DMA-åtkomst skall göras. Tar som argument nuvarande tidpunkt mätt i maskincykler refererat till grafikprocessorn.

void SaveSpriteData(const unsigned int)  
 - Sparar data om grafisk information för de rörliga grafiska objekten. Tar som argument nuvarande rad under utritning på skärmen refererat till grafikprocessorn. Informationen sparas i en lista av listor av typen ULONG som en global variabel vid namn spriteDmaData.

void ReadGfxDmaData(const int)  
 samt  
 - Sparar data om grafisk information angående vilken typ av tecken färg som skall visas på skärmen, enligt vad grafikprocessorn läser från minnet. Tar som argument den kolumn refererat till grafikprocessorn som DMA-åtkomsten har påbörjats. Informationen sparas i en lista av heltal som en global variabel vid namn charDmaData.

void ReadGfxRowData()  
 - Sparar data om grafisk information enligt vad grafikprocessorn läser från minnet för varje rad på skärmen. Informationen sparas i en lista av listor av heltal som en global variabel vid namn gfxDmaData.

Följande funktioner är interna underfunktioner som anropas av funktionen Run6510.

void FixEvents(unsigned int&)  
 - Kontrollerar om grafikprocessorn har passerat tiden för utritning av en hel skärmsida, och i så fall anpassar alla händelser och efter detta. Som argument tar funktionen nuvarande tidpunkt mätt i maskincykler refererat till grafikprocessorn.

Blocket VIC utgörs av en klass C64 och sköter generering av grafisk information. Dock så måste det ha tillgång till blocket EMULATOR med dess definitioner. Definitioner av blocket VIC finns i filen emulator.h, samt programdelen finns i filerna vic.cpp och vic\_defs.h.

För extern kommunikation har klassen C64 följande funktioner.

unsigned int\* UpdateNextRaster()  
 för  
 - Genererar nästa rad av grafisk information från grafikprocessorn. Raden returneras som en lista av 504 heltal, där varje heltal är färgen den punkt på raden som motsvaras av positionen i listan. Som grundinformation använder sig funktionen av stackarna i de globala variablerna gfxsavedata, spriteSaveData, charDmaData och gfxDmaData.

Följande funktioner är interna underfunktioner som anropas av funktionen UpdateNextRaster.

unsigned int UpdateScreenEvent(unsigned int)  
 - Hanterar händelser specifika för bakgrundsgrafik. Tar som argument och returnerar antalet maskincykler som återstår tills nästa händelse angående bakgrundsgrafik.

unsigned int UpdateSpriteEvent(unsigned int spritecy,unsigned int mask)  
 - Hanterar händelser specifika för rörliga grafiska objekt. Tar som argument spritecy och returnerar antalet maskkcykler som återstår tills nästa händelse angående rörliga grafiska objekt. Argumentet mask beskriver vilka rörligt grafiskt objekt som berörs genom motsvarande bit satt till aktiv (1) eller inaktiv (0).

## 11.1. Systemdokumentation PC

Här beskrivs huvudprogrammet för implementering för PC med MS-DOS. Definitioner och programdel av huvudprogrammet finns i filerna main.cpp och fixbios.asm.

Programmet består huvudsakligen av följande funktion.

main()  
 från  
 avbryts.  
 - Tolkar kommandoraden och läser in alla nödvändiga filer som behövs för att köra emuleringen. Initierar allt för att kunna starta emuleringen. Sedan körs emuleringen genom anrop till emulator kärnan, inläsning tangentbord och utritning av grafik på skärmen tills programmet avbryts.

Följande funktioner är underfunktioner som anropas av main.

void addbios()  
 Hz.  
 void rembios()  
 innan  
 int readtimer()  
 Hz.  
 int readrawkey()  
 void initgfx(int)  
 void restoregfx()  
 void drawgfx(unsigned int \*gfx,unsigned int dest,unsigned int len,unsigned int mode)  
 argumentet  
 Om

- Startar igång en avbrottshanterare för tangentbordet samt en avbrottshanterare för en timer som är satt ge avbrotts signaler med 50 Hz.  
 - Återställer de ursprungliga avbrottshanterarna som var aktiva innan anropet till addbios.  
 - Returnerar hur många avbrott som getts från timern som är satt till 50 Hz.  
 - Returnerar den tangentbordskod som har rapporterats från tangentbordets hårdvara, returnerar noll om inget har rapporterats.  
 - Initierar grafikkortet till det grafikläge som specificeras av argumentet.  
 - Återställer grafikkortet till det grafikläge som var aktivt innan anropet till initgfx.  
 - Ritar ut grafik på skärmen med från en lista av heltal enligt gfx som innehåller antalet färgpunkter enligt argumentet len, till grafikkortets minne från och med adress enligt argumentet dest.  
 Om argumentet mode är noll används så kallat linjärt läge för utritningen. Annars används argumentet mode som värde för subtraktion per utritningsfas vid så kallat planar läge.

## 11.2. Systemdokumentation UNIX/X

Här beskrivs huvudprogrammet för implementering för UNIX med X-Windows. Definitioner och programdel av huvudprogrammet finns i filen main.cpp.

main() - Tolkar kommandoraden och läser in alla nödvändiga filer som behövs för att köra emuleringen. Initierar allt för att kunna starta emuleringen. Sedan körs emuleringen genom anrop till emulator kärnan, inläsning från tangentbord och utritning av grafik på skärmen tills programmet avbryts.

Följande funktioner är underfunktioner som anropas av main.

void OpenC64Window()  
- Öppnar ett fönster i X-Windows för utritning av grafik.

void CloseC64Window()  
- Stänger fönstret.

void CreateImage() - Skapar en representation av skärmen för C64 i minnet.

void DrawGraphics(unsigned int)  
- Ritar ut grafik i fönstret i antalet rader enligt argument.

int ParseEnvironment(char \* name,char \* var,char \*\* envp)  
- Tolkar miljövariablerna enligt argumentet envp. Letar efter miljövariabeln med namn enligt argumentet name, och tilldelar argumentet var innehållet i miljövariabeln.

void ParseOptions(int argc,char \*\* argv)  
- Tolkar kommandoraden. Argumentet argc specificerar hur många olika texter som står på kommandoraden och argumentet argv är en lista av dessa texter.

## **12. Referenser**

Commodore Business Machines, 1982, *Commodore 64 Programmer's Reference Guide*, Howard W. Sams & Co.

Marko Mäkelä med flera, 1994, *Documentation for NMOS 65xx/85xx Instruction Set*, USENET

Marko Mäkelä med flera, 1994, *The memory accesses of the MOS 6569 VIC-II*, USENET

### **13. Ordlista**

BCD	Förkortning för Binary Coded Decimal. Detta är ett system att koda decimala värden i ett binärt minne. Förenklat går det ut på att varje siffra i det decimala talet representeras av 4 bitar.
DMA	Förkortning för Dynamic Memory Access. Innebär att andra enheter förutom huvudprocessorn i ett datorsystem kan göra åtkomst till primärminnet. Oftast tvingas huvudprocessorn att stanna exekveringen då detta försiggår.
MMU	Förkortning för Memory Management Unit. Detta är en krets eller logik som hanterar åtkomsten till primärminnet. Oftast är den styrbar och används till att dirigera om olika adresser till olika minneskretsar eller enheter.
MS-DOS	Ett operativsystem som används på en PC och är tillverkat av företaget Microsoft. Detta operativsystem är föråldrat och har många nackdelar som att endast 640 kbyte av primärminnet kan utnyttjas på ett effektivt sätt.
PC	Förkortning för Personal Computer. Denna beteckning används nästan uteslutande för benämning av datorer som är kompatibla med företaget IBM's datorer.
UNIX	Beteckning för en samling av operativsystem avsedda för arbetsstationer och stordatorer. Dessa operativsystem har ett standardiserat gränssnitt gentemot applikationerna och bygger på användning av programspråket C. Operativsystemen är oberoende av processortyp och datorarkitektur då samtliga program som är konstruerade i C för UNIX går att flytta och kompilera under ett annat operativsystem för UNIX.
VGA	Förkortning för Video array Graphics Adapter. Detta är ett grafikkort som används till en PC och möjliggör visning av upp till 256 färger i en maximal upplösning av 640 gånger 480 bildpunkter på en skärm.
X-Windows	Beteckning för ett grafiskt användargränssnitt avsett för UNIX som presenterar informationen grafiskt i olika oberoende områden på skärmen, så kallade fönster.