Correct Emulation of Micro-Computer C64 in Software independent of Computer System

CCS64 - A Commodore 64 Emulator

A Diploma Work in Computing Science

HÅKAN SUNDELL

DEPARTMENT OF COMPUTING SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 1996
www.chalmers.se

Diploma work 1996

Correct Emulation of Micro-Computer C64 in Software Independent of Computer System

CCS64 - A Commodore 64 Emulator

HÅKAN SUNDELL

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

Department of Computing Science
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden 1996

Correct Emulation of Micro-Computer C64 in Software Independent of Computer System
CCS64 - A Commodore 64 Emulator
HÅKAN SUNDELL

Supervisor: Christer Bernérus, Department of Computing Science
Examiner: Kent Petersson, Department of Computing Science

Diploma Work 1996
Department of Computing Science
School of Mathematical and Computing Sciences
Chalmers University of Technology | University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Official icon of the CCS64 executable for Windows.

Typeset in LaTeX
Translated from Swedish into English in 2024
Gothenburg, Sweden 1996, 2024

Correct Emulation of Micro-Computer C64 in Software independent of Computer System
CCS64 - A Commodore 64 Emulator
HÅKAN SUNDELL
Department of Computing Science
Chalmers University of Technology | University of Gothenburg

Abstract

The purpose with this work has been to explore if correct emulation of the micro-computer 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.

Keywords: commodore 64, emulation, micro-computer, c++.

Acknowledgements

This is a diploma work of 20 credits in Computing Science at the Department of Computing Science, Chalmers University of Technology / University of Gothenburg.
The diploma work is part of a 1 year's Master's Degree in Computing Science.
The work was carried out in 1995 with about 800 working hours.
The supervisor has been Christer Bernérus.
The examiner has been Kent Petersson.

Håkan Sundell, Gothenburg, April 1996

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

BCD	Short for Binary Coded Decimal. This is a system of encoding decimal values in a binary memory. In simple terms, each digit in the decimal number is represented by 4 bits.
DMA	Abbreviation for Dynamic Memory Access. Allows other devices besides the main processor in a computer system to access primary memory. Most often, the main processor is forced to stop execution when this is taking place.
MMU	Abbreviation for Memory Management Unit. This is a circuit or logic that manages access to primary memory. Most often, it is controllable and is used to redirect different addresses to different memory circuits or devices.
MS-DOS	An operating system used on a PC and manufactured by the company Microsoft. This operating system is outdated and has many disadvantages such as the fact that only 640 kbytes of primary memory can be utilized efficiently.
PC	Abbreviation for Personal Computer. This designation is used almost exclusively to designate computers that are compatible with IBM's computers.
UNIX	A collection of operating systems designed for workstations and mainframes. These operating systems have a standardized interface to the applications and are based on the use of the C programming language. The operating systems are independent of processor type and computer architecture, as all programs designed in C for UNIX can be moved and compiled under another operating system for UNIX.
VGA	Short for Video array Graphics Adapter. This is a graphics card used for a PC and allows the display of up to 256 colors in a maximum resolution of 640 by 480 pixels on one screen.
X-Windows	Designation for a graphical user interface intended for UNIX that presents the information graphically in various independent areas of the screen, known as windows.

Contents

1

Introduction

1.1 Background

The Commodore 64 (C64) is a microcomputer that was very popular during the years 1982 - about 1987. It has been produced in over 10 million copies, which have been sold mainly in the Western world. In recent years, sales are made to another market, especially to Eastern Europe. The computers were mainly used for computer games, but there could also be limited computer work. There are a very large number of commercial programs for the C64, more than 10 000, and some new production is still taking place.

With today's personal computers being much more powerful, few people still own or use the C64 in the Western world. However, there is still a lot of interest in being able to use the software available for the C64. Therefore, there are a number of so-called emulators on the market, both commercial and free. These are programs written for a particular computer system, which enable the use of software written for another computer system. However, these have in common that only a fraction of all software for C64 can be used, i.e. they do not work properly.

One reason for this is that they are interpretive in real time and try to synchronize the execution of the C64 program with the real time flow. Different parts of the C64 program take different amounts of time to interpret, which interferes with synchronization. If you are to succeed with this emulator principle, you have to use an extremely powerful computer system where it takes the C64 program part that is interpreted the slowest as if you had executed it on a real C64.

Another reason for the difficulties with proper emulation is the lack of information available, as well as too little effort on research.

1.2 Existing solutions

There are systems that emulate C64 for computer environments such as PC, Amiga and Unix/X:

- **C64S**. This system is written entirely in assembler and runs under the operating system MS-DOS. It's completely tied to the existing hardware in a PC. To reach the same speed as a real C64, an Intel 486/66 Mhz processor or better is required. The speed of updating the information on the screen is optional and occurs at set time intervals. Advanced applications that require total synchronization between the processor and the other circuits result in some distorted graphical information on the screen, as well as the program is executed incorrectly and can often cause so-called locking, also of the emulator itself. The system is commercial and costs about SEK 500.

- **PC64**. This system is broadly similar to the C64S but can show a little better accuracy. The increase in accuracy has been made with the help of a number of parameters. These are adjusted by hand so that a particular program is executed more or less correctly. This system is also commercial. The manufacturer provides a list of parameters suitable for different applications.
- **X64**. This system is written entirely in C and runs under the UNIX/X-Windows operating system. Any program that requires any synchronization between the processor and the other circuits is executed completely incorrectly. This means that only a small number of programs are useful. The system was made as a degree project in 1991 and is completely free.
- **A64**. This system is written entirely in Assembler and runs under the AmigaDOS operating system. It's completely tied to the existing hardware in an Amiga. To reach the same speed as a real C64, a Motorola 68030 / 50 Mhz processor or better is required. The system shows slightly better accuracy with programs that require synchronization than X64. The system is commercial.

1.3 Target and aim

The purpose of my work is to develop a new *emulator principle*, specifically designed for emulation of C64. This emulator principle should lead to implementations that have such good correctness that almost all programs intended for C64 are executed correctly. There is more emphasis on the C64 program being executed correctly than on communication with the outside world being completely correct or synchronous. That is, the C64 program should be executed in a virtual world where all events are perceived internally as synchronous, while externally they may well be perceived as asynchronous.

2
Theory

2.1 Information gathering

In order to develop a principle for the emulation, I need to have a correct knowledge of how the microcomputer C64 works.

A C64 can be divided into these basic units:

- Keyboard
- Circuit Board
- Interfaces

The printed circuit board consists mainly of these devices:

- Processor (6510)
- Memory (64k RAM, 20k ROM)
- Timer/In/Out Unit (6526)
- Graphics Processor (VIC 6569)
- Sound Processor (SID 6581)
- Memory Manager (MMU)
- Address/Data Bus (16/8 bits)

See Figure 2.1 for a simplified block diagram of the circuit board.

The C64 I have studied and intend to emulate is for the PAL system. This means that it is adapted for the European TV system PAL based on 312 lines updated with 50 Hz or 625 lines interlaced. The equivalent system in the US is NTSC which is based on 262 lines updated with 60 Hz or 525 lines interlaced.

The address and data bus are clocked at 985248 Hz (1022727 Hz NTSC) and it is also at this speed that the processor and the other circuits work.

The processor, a MOS6510, is actually a MOS6502 with a built-in 8-bit in/out register. The address bus is 16-bit and the data bus 8-bit, which means an addressable space of 64 kbytes. There are 5 accessable 8-bit registers, status SR, stack pointer SP, accumulator A and indexes X and Y. Primarily, the accumulator is connected to the arithmetical and logical unit (ALU) that is able to perform various bit-wise operations as well as basic mathematical operations on values consisting of 8 bits with carry. These values are either interpreted as an unsigned, signed, or a Binary Coded Decimal (BCD) number. To access internal and external in/out registers, so-called memory mapped input/output is used. This means that these registers

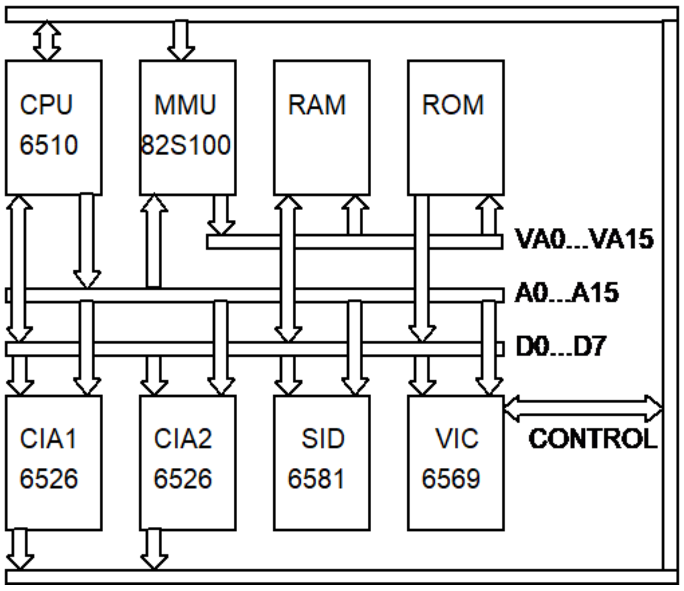


Figure 2.1: A simplified block diagram of the Commodore 64 circuit board.

are seen by the processor as any memory cell. The data format of the 16-bit values in memory is the least significant bytes first. Officially, there are 56 different instructions and the number of addressing methods is 13. There are two types of interrupts, one maskable (IRQ) and one non-maskable (NMI).

To be able to address more than 64 kbytes, there is a memory management unit (MMU). This is partly controlled by the internal register of the processor. The addressing can therefore switch between, among other things, 64 kbytes of RAM, 20 kbytes of ROM and 4 kbytes of I/O.

There are two timer/in/out units of type MOS6526, named as Complex Interface Adapter (CIA). They each contain two 8-bit in/out registers, two 16-bit timers, a clock, a shift register and a maskable interruption register. In total, there are 16 8-bit registers per circuit that can be accessed through the bus. These circuits can generate both types of interrupts that are handled by the processor.

The graphics processor is of type MOS6569, named as Video Interface Chip (VIC), and generates a video signal with a horizontal frequency of 15625 Hz and a vertical frequency of 50 Hz (for PAL systems). The resolution is constant, about 384 dots per horizontal line are visible on a regular TV. There are various different selectable text and graphics modes, 40 by 25 characters of text or graphics with 320 by 200 and 160 by 200 pixels, respectively. The text or graphics can be shifted in height and sideways. In addition to this, there are also 8 freely moving objects that can overlay the text or graphics, so-called Sprites. The number of colors that can be chosen from is 16, but with some limitations regarding individual pixels. This circuit can also generate interrupt signals, in this case maskable (IRQ). In total, there are 31 8-bit registers that can be accessed through the bus. To retrieve various information from memory about which characters or graphics are to be drawn on the screen, so-called Direct Memory Access (DMA) is used. This means that the graphics processor can temporarily prevent the processor from accessing the memory, as it uses the bus to access the memory itself.

The sound processor is of the MOS6581 type, named as Sound Interface Device (SID), and generates single-channel sound. Internally, there are three sound generators that are mixed together to a sound that is set to volume with the main volume register. For each sound generator, there is a waveform generator that can generate triangle, sawtooth, square, and noise waveforms in frequencies ranging from 30 Hz to 12 kHz. In addition, there is a programmable amplitude generator for each sound generator, a so-called Envelope, with the help of which you get different timings to imitate real instruments. These sounds from the three sound generators can optionally be mixed together in different ways. A programmable filter can also be applied to the mixed audio. In addition to audio functions, there are also two 8-bit analogue to digital converters. In total, there are 25 8-bit registers that can be accessed through the bus.

A lot of information about the processor and the other circuits is documented in books such as the Commodore 64 Programmer's Reference Guide [1]. However, there is various necessary information that has not been documented so far. Therefore, I have been forced to research this information on my own. Since I have not had access to sophisticated electronic measuring equipment such as a logic analyzer, I have instead tried to write special assembly programs and have been able to draw various conclusions from their execution results.

The circuits that I have researched more closely are the processor, the timer/in/output units and the graphics processor. In addition to my own research, I have verified documented information [1] [2] [3] and researched new information when the documented information has sometimes been incorrect.

What has been unclear about the processor is the interruption handling, bus handling and how undefined BCD values and instructions are handled. Regarding the interruption management, I wanted to know exactly how many machine cycles an interrupt sequence takes, how priorities between maskable and non-maskable interrupt are managed, and exactly when and on what criteria an interrupt sequence is executed. Bus handling refers to how and what the processor addresses during the execution of a certain instruction, in fact it is studied how the so-called nano-code in the processor works. Since an 8-bit BCD value is only defined for about half of the values an 8-bit register can assume, I want to investigate how arithmetic calculations with undefined BCD values are performed. An instruction word to the processor is 8-bit, but only 56 of these values are defined as instructions. Therefore, I want to investigate what is being done by the processor if you try to use some undefined value as an instructional word.

The timer/in/out device, as mentioned earlier, has two 8-bit in/out ports. I want to investigate how these are interpreted internally when, for example, a high level is connected to a port that is set as an entrance. Furthermore, I want to investigate what happens when you change the counting method for a timer.

The graphics processor works completely synchronously with the main processor. I want to know exactly how many machine cycles correspond to a horizontal line and an entire drawing of the screen image. Because the main processor can force the graphics processor to change text or graphics mode at any time during the drawing of the screen, various effects can occur. Therefore, I want to investigate what happens when you change one of the graphics processor's registers and also exactly when the changes are visible on the screen. Furthermore, I want to study how the graphics processor handles DMA access, exactly when it takes place and under what criteria.

2.2 Processor

See Figure 2.2 for a block diagram of the MOS 6510 microprocessor.

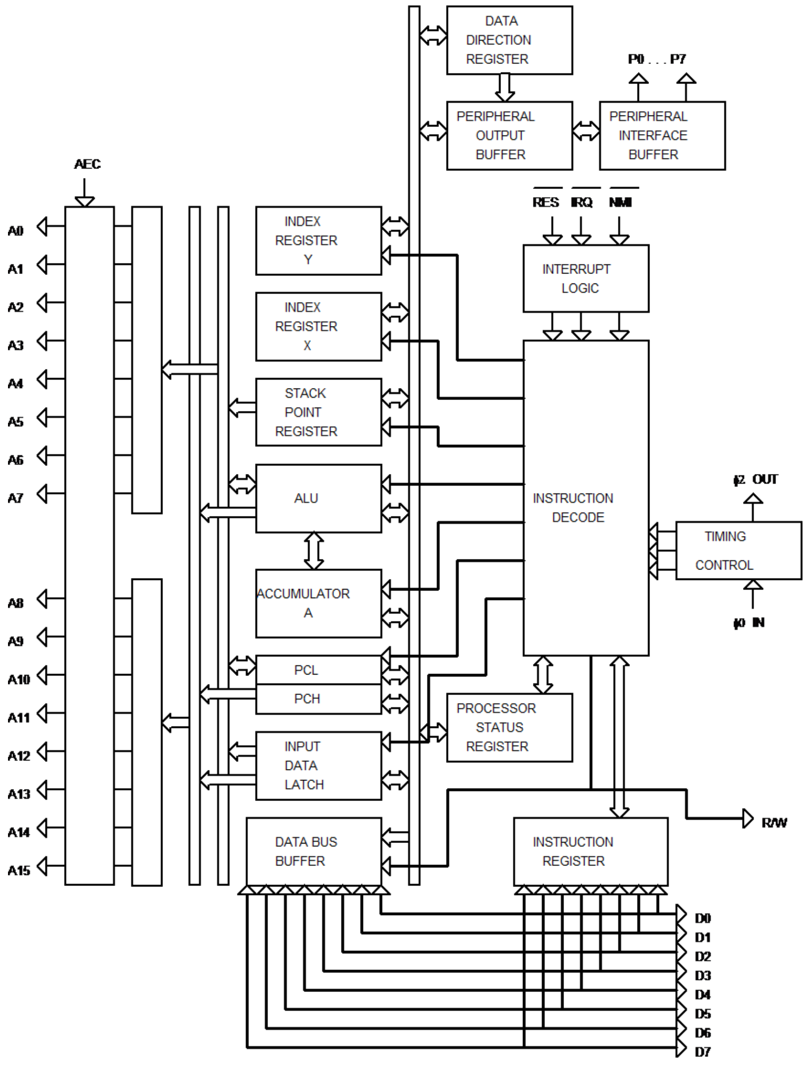


Figure 2.2: A block diagram of the MOS 65010 microprocessor.

To investigate how undefined BCD values are handled in arithmetic operations such as addition and subtraction, I have used a so-called machine code monitor on C64. With the help of this program, I have been able to execute sequences of assembler code and then be able to check the status of registers, flags and memory before and after execution. The program for checking addition with BCD values looks like this.

```
SED; Select arithmetic mode to manage BCD.
CLC or SEC; Resets or sets the C (Carry) memory flag.
```

```
LDA #value1 ; Setting the accumulator to value1
ADC #value2 ; Adding value2 to the accumulator
```

The answer of the operation is given in the accumulator, in addition the flags Z (Zero), N (Negative), V (Overflow) and C (Carry) are set depending on the result. In order to draw some conclusions from how the handling of undefined BCD values is handled, I have tested with different values of value1 and value2 respectively and checked the results. I have then entered these test results into a table, after which I have analyzed this and come up with a plausible algorithm. To test the equivalent for subtraction, I have replaced `ADC` with `SBC` in the program.

The analysis of undefined instructions has been carried out in a similar way. Here, however, I have not been able to write in assembler but had to write the program as machine code in hexadecimal values.

To test the interruption management, I have used different programs that force maskable and non-maskable interruptions at different times, in order to be able to analyze how the prioritization between these interruptions is managed. The information base I have worked from is from the document regarding the 6510 processor by Marko Mäkelä and others [2]. However, this document has several inaccuracies such as regarding the interruption management, which is why I also had to verify all the information in this document in order not to base my research on incorrect grounds.

2.3 Information gathering regarding timer/in/out devices

Here too I have used a machine code monitor and various small assembly programs. One of the programs looks like the following below. It tests changing the working mode of Timer A.

```
LDA #value1
STA $DC04 ; Set the minimum significant changes of the starting
    value of Timer A.
LDA #value2
STA $DC05 ; Set the maximum significant changes of the starting
    value of Timer A.
LDA #value3
STA $DC0E ; Set Timer A working mode to value3
<Time Delay (Empty Loop)>
LDA $DC04 ; Read the instantaneous value of the least significant
    changes of Timer A.
STA save1 ; Save this value in memory address save1.
LDA #value4
STA $DC0E ; Change the working mode of Timer A to value4
LDA $DC04 ; Read the instantaneous value of the least significant
    changes of Timer A
STA save2 ; Save this value in memory address save2.
```

To test how in/out ports are interpreted internally and externally, I have set the in/out registers to various values, set the configuration for the ports to different combinations of inputs and outputs. After that, I have externally imposed and read different levels.

2.4 Graphics processor information gathering

Here too I have used a machine code monitor and various small assembly programs. In order to test what happens to the drawing of the screen image when you change any of the graphics processor's registers, I have had to develop a program that is completely synchronized with the drawing on the screen. This program looks very simplified as follows.

```
LDA $D012 ; Read which line is currently plotted on the screen
CMP value-1 ; Does this equal value-1 ?
BNE -2 ; If no, read the line again, etc.
LDA value
STA $D012 ; Set the register that compares and causes interrupts
    when the set value equals the current row.
NOP; Do nothing in two machine cycles.
NOP
...; And so on...
```

- After a while, an interrupt will be generated and the application flow will change to

```
LDX #09
DEX
BNE -2 ; Executes an empty loop 9 times.
NOP
NOP
LDA $D012 ; Read which line is currently being drawn.
CMP value+1 ; Does this equal value+1
BNE 0 ; If not, do nothing in a machine cycle.
```

- Here we are exactly at Y=value+1 and X=5.

Then the program is executed that sets the respective registers to different values to be examined. The register that has included a great deal of research and analysis is the register (namely, register 17) that controls the DMA's access to memory for graphics and textual information. For this register, I have managed to figure out the algorithm that the graphics processor uses for this register. Before my work, this register's impact on DMA access has been almost to be regarded as random [3].

2.5 Results from information gathering regarding processor

External interrupts are executed as the next instruction if the interruption is signaled to the processor and is active no later than the penultimate machine cycle of the execution of an instruction. All interruptions take 7 machine cycles. The variations are executed as follows in Tables 2.1, 2.2, 2.3 and 2.4.

Table 2.1: BRK (instruction, forced stop)

#	Address	R/W	Description
1	PC	R	reads code for the operation, increase PC by one.
2	PC	R	reads and throws away, increasing PC by one.
3	0x0100+S	W	writes bits 8 to 15 of PC, then reduce S by one.
4	0x0100+S	W	writes bits 0 to 7 of PC, then reduce S by one.
5	0x0100+S	W	writes the status register with B set to one, set I to one, then reduce S by one, if NMI is active, the next step is executed as NMI otherwise as IRQ.

Table 2.2: IRQ and NMI

#	Address	R/W	Description
1	PC	R	reads and throws away
2	PC	R	reads and throws away
3	0x0100+S	W	writes bits 8 to 15 of PC, then reduce S by one.
4	0x0100+S	W	writes bits 0 to 7 of PC, then reduce S by one.
5	0x0100+S	W	writes the status register with B set to zero, set I to one, then reduce S by one, if NMI is active, the next step is executed as NMI.

Table 2.3: IRQ and BRK

#	Address	R/W	Description
6	0xFFFE	R	reads bits 0 to 7 of PC.
7	0xFFFF	R	reads bits 8 to 15 of PC.

Table 2.4: NMI

#	Address	R/W	Description
6	0xFFFA	R	reads bits 0 to 7 of PC
7	0xFFFB	R	reads bits 8 to 15 of PC

Non-maskable interruption (NMI) is interpreted as active only when the NMI has been re-signalled. However, normal interruption (IRQ) is always interpreted as active when IRQ is signaled and I is set to zero.

For the handling of BCD values, algorithms have been developed for both addition and subtraction. All undefined instructions have provided algorithms that are verified.

2.6 Results from information gathering regarding timer/in/output device

Changing the working mode of any timer, i.e. writing to register 14 or 15, is done as follows in Tables 2.5, 2.6, 2.7 and 2.8.

Table 2.5: Write to 6526 in order to stop timer.

#	R/W	Data	Description
0	W	xxx0xxx0	Write to 6256
1	-	-	register = data
2	-	-	Stop timer

Table 2.6: Write to 6526 in order to start timer

#	R/W	Data	Description
0	W	xxx0xxx1	Write to 6526
1	-	-	register = data
2	-	-	Start timer, reduce timer by one

Table 2.7: Write to 6526 in order to reset timer from latch and stop timer.

#	R/W	Data	Description
0	W	xxx1xxx0	Write to 6256
1	-	-	register = data
2	-	-	timer = latch, Stop timer

Table 2.8: Write to 6526 in order to reset timer from latch and start timer

#	R/W	Data	Description
0	W	xxx1xxx1	Write to 6526
1	-	-	register = data
2	-	-	timer = latch
3	-	-	Start timer, reduce timer by one

The input/output ports are interpreted the same internally (reading of records) as externally as follows:

$$value = (outgoing | \sim control) \& inbound \tag{2.1}$$

2.7 Graphics processor information gathering results

The time for drawing a full screen page (312 lines - PAL) corresponds to 19656 machine cycles.

Any register regarding colors provides change to the screen immediately. Registers 22, 24 and bits 4 through 6 of registers 17 provide change with a lag of one machine cycle.

The management of the DMA access works according to the following algorithm expressed as C-like pseudo-code.

```
if(register17[bit 0-2] == lineCounter[bit 0-2] )
    graphics = 1;

if(register17[bit 0-2] = lineCounter[bit 0-2] and 12 <
   columnCounter < 45)
    dma = 1;
else
    dma = 0;

if(columnCounter == 14 and dma == 1)
    lineCounter = 0;

if(register17[bit 0-2] == lineCounter[bit 0-2] and 15 <
   columnCounter < 45)
{
    characterBuffer[columnCounter-15] = memory[characterAddress];
    colorBuffer[columnCounter-15] = memory[colorAddress];
}

if(graphics == 1 and 15 < columnCounter < 45)
{
    characterAddress = characterAddress + 1;
    colorAddress := colorAddress + 1;
}

if(columnCounter == 45 and graphics == 1)
{
    if(lineCounter < 7)
    {
        lineCounter = lineCounter + 1;
        characterAddress = characterAddress - 40;
        colorAddress := colorAddress - 40;
    }
    else
        graphics = 0;
}
```

`lineCounter` - The raster line that is currently being drawn on the screen.

`columnCounter` - The column that is currently being drawn. There are 63 columns per line. The columns correspond exactly to the machine cycles in terms of time.

`dma` - If active, the bus is owned by the graphics processor.

`graphics` - If active, graphical information is drawn in this column.

`lineCounter` - The line (measured in points) of the character that is being drawn.

`characterBuffer` - This buffer contains information about which characters should be drawn.

`colorBuffer` - This buffer contains information about which color to draw.

`characterAddress` - Index to the memory address from which information about characters is to be read.

`colorAddress` - Index to the memory address from which information about colors is to be read.

For the first three machine cycles with DMA active, there is no actual read from the bus, as the graphics processor has to wait for the main processor to finish accessing the bus. If the graphics processor tries to read from memory during these first three machine cycles, the value will be undefined, however, it is usually hexadecimal FF.

With the right programming of the DMA management, you can achieve effects such as moving the information in the graphic area in both the x and y directions, as well as increased graphic quality with more colors per unit area.

Furthermore, results have been achieved in the following areas.

- The management of the screen blanking. If you change the criteria for this at the right time, the solid color border around the graphic area of the screen can be turned off, allowing graphic information to be visible outside the original graphics area.
- The handling of moving objects. With the right programming of these moving objects, you can achieve more than eight visible moving objects, as well as an extra increase in the height of the objects.

3 Emulation

The first thing that needs to be decided is what is to be emulated. External peripherals I have excluded almost completely with the exception of the TV screen, keyboard and joystick. However, it is important that the emulator gets such an open architecture that the addition of support for various external devices is possible without major changes. Furthermore, emulation of the sound processor is excluded for the time being as it is difficult to find a standard for sound that is supported by all modern computer systems.

The basic idea of the emulator is that it should consist of a core, an object in C++. This object should be completely independent of computer architecture. All communication between the outside world, i.e. the screen, keyboard and joystick, takes place according to fixed protocols between the core and the main program. The main program is thus a customization program that handles the problems specific to the computer architecture for which the emulator is to be used. However, the core has an internal representation of the screen, keyboard, and more.

The parts that the core will emulate are broadly as follows:

- Processor.
- Graphics processor including screen image generation.
- Memory with memory manager.
- Timer/In/Out devices with parts of interfaces.
- The bus with DMA and more.

The computer systems for which the emulator is supposed to work are mainly systems with one processor, which is why parallel programming should be impossible or otherwise very inefficient. Since I have not chosen parallel but instead sequential programming, I have to divide the parallel processes that actually take place in C64 sequentially. The completely dominant work in the computer is carried out by the processor and therefore it also has a dominant position in the course of the emulator. The other units are emulated so that only urgent work is performed directly while less urgent work is piled up and performed later in larger work blocks.

For the timer/in/out unit, all work is of an urgent nature, and therefore it must be emulated so that it is handled immediately when needed. The same goes for the memory manager and memory. The graphics processor's work consists of both emergency and non-emergency parts, which is why the emulation has to be divided into two parts, one for emergency and one for non-emergency work.

The construction of the emulator begins with the processor, whose design and principles set guidelines for how the remaining parts of the emulator should be constructed.

3.1 Processor emulation

I have done the emulation of the processor as an object in C++. This item also includes a portion of the memory and memory manager. The first difficulty is to reason out how the processor's register should be emulated. The records available are as follows:

- **PC** - Program counter, a 16 bit register. This register determines the address in the memory from which the instructional words are to be retrieved.
- **A** - Accumulator, an 8 bit register. It is with this register that almost all arithmetic operations work.
- **X** – Index, an 8 bit register. This register is used to index addresses.
- **Y** – Index, an 8 bit register. This register is used to index addresses.
- **SP** – Stack pointer, an 8 bit register. This register is used to keep track of which address the host should be saved or retrieved on the stack.
- **SR** – Status register, an 8 bit register. This record is used to keep track of how recent arithmetic operations have progressed. The things that are recorded are if the result is equal to zero (Zero), if it is negative (Negative), if it became too large (Carry) or if it became too large so that it is not possible to determine whether it is negative or not (oVerflow). In addition, it is recorded whether numbers should be handled as BCD values in arithmetic operations (Decimal), if maskable interrupt is allowed (Interrupt) and if we have executed a BRK instruction (Break).

The registers PC,A,X,Y and SP can clearly be emulated as variables in C++. As for SR, it is very inefficient to construct SR after each arithmetic operation as each flag (Z,N,C,V,D,I,B) is represented as a piece in SR. In addition, it is very rare that all flags are affected by an operation. Therefore, each flag is emulated as its own variable in C++, with the exception of Z and N, which are instead emulated with a common variable. This is most effective when Z and N are set at the same time. Since these flags are always based on an 8-bit result, this result is stored in the ZN variable instead of the Z and N flags that can be derived if necessary. Yet another reason to emulate each flag as a single variable is that conditional jump instructions are always based on the state of a flag, so do many arithmetic operations. However, the entire SR can be retrofitted if necessary.

To emulate the processor's execution of instructions, it is very suitable with the switch-case statement in C++. This is very suitable as each instruction consists of one byte with zero to two bytes as operands. This makes for a switch statement with 256 different case statements.

The instruction execution should continue until an interruption occurs, so a while clause can be used with great advantage as shown in the following diagram.

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

When an interruption occurs, the while loop is interrupted and the interruption is handled, after which the while loop is resumed. For the emulated processor, however, there are more exceptions than interruptions to be handled. An exception is, for example, that we want to interrupt the execution of instructions. Therefore, the sketch is modified as follows.

```
while(!exception)
{
  ...
}
if(interrupt) ...
else if(stop) ...
...
```

The emulation of the instructions themselves is very strict in relation to how the processor executes them in reality. This means that the instructions are emulated in similar steps to those carried out internally in the processor. Since many execution steps are very similar, it is advisable to introduce subroutines or even better so-called macros. Macro refers to abbreviations for larger program statements. Here is an example of emulation of instruction A9.

```
case 0xa9: /* LDA Immediate - That is, read the byte that
follows the instruction word and place this value in the
accumulator, then the Z and N flags are set depending on
this value. */
a = READFROMMEMORY(pc++);        /* A is assigned the value
from memory in address PC, then PC is increased by one. */
zn = a;          /* flags Z and N are assigned the value in A */
break;
```

The next difficulty is to reason out how the emulation of memory management should be done. A good way is to type the memory. As mentioned earlier, there are at least three types of memory, RAM, ROM and I/O. For each memory address, you can have a corresponding value in a table that describes the memory type. However, the memory management can be changed as the program is executed, which is why you have to have a table of different tables of memory type. These values that represent the memory type can also represent more information. In our case, there are only about 100 available I/O registers, which is why we can let the memory type also represent which register is being referred to. This can be effective as the same I/O register can be accessed through different memory addresses. Therefore, a read from memory can be emulated as follows.

```
reg = MemoryTable[adress];
```

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

However, it is not only the processor that can read and write to the memory, but also other devices such as the graphics processor can access the memory through DMA. Therefore, we must check whether such a thing should be done before we make reading or writing in memory. Since these DMA accesses occur in sync with the processor's execution, we need to introduce a variable that counts the number of machine cycles that have elapsed since the start of the processor. In addition, not only does DMA occur synchronously with the processor, but other devices can also change the value of any I/O register or generate interruptions to the processor at a certain time. Then there is the fact that DMA also prevents the processor from doing any execution during a certain number of machine cycles. So the access control looks like this.

```
machineCycle++;                    /* Next machine cycle */
if(machineCycle == eventCycle)
{
    machineCycle = CheckEvent(machineCycle);
    eventCycle = NextEvent(eventCycle);
}
```

3.2 Emulation of bus handling and more

This part emulates what occurs when the processor wants to make an access or access control to the bus with the possible handling of some event. Furthermore, it can be said that it emulates the emergency work to be done for any of the other units.

Since the object that emulates the processor also handles a certain part of the access to the memory, the emulation of the access to the I/O registers of the various devices remains. Here are the main features of the emulation for reading the I/O registers.

```
switch(register)
{
  case 1: /* For example, the graphics processor, background color
      */
    return backgroundColor;
  case 2:
    return ...
  ...
}
```

When writing to various I/O registers, it may happen that non-urgent work arises that can be carried out later. Then this is solved by using different stacks. On these stacks, information is saved about which register was written to, what was written and when (which machine cycle). It is advisable to have different stacks for different devices, so that when the emulation for this non-urgent work for a particular device is executed, only one stack has to retrieve information from.

Incidents are handled in a similar way, as these can also result in non-emergency work. Then information about this work is also saved on stacks. Often this non-urgent work is of the nature that it does not have to be performed in order for the processor to execute the program correctly, but only to provide information to the outside world, such as the screen.

To make the emulation of events effective, these are divided into different areas, including the device they originate from. The emulation of the event handling is as follows.

```
switch(basicType)
{
    case 1: /* For example, that some device makes DMA access */
      <stop the processor for x number of machine cycles>
      break;
    case 2: /* Graphics processor */
      switch(specType)
      {
        case 1: /* generate maskable interrupt IRQ */
          ...
          break;
        case 2:
          ...
      }
    case 3:
     ...
}
```

The basic types of events that are handled are stop, DMA, exception, graphics processor, incoming data, timer 1 and timer 2.

Why it is appropriate to divide the events according to this is that we have to sort the events according to which machine cycle the event should occur every time a new event occurs. The event that occurs closest to the machine cycle that the processor is currently executing is the event whose machine cycle should be compared each time the processor performs an access check. Generation of new events can come from, among other things, writing or reading I/O records, or as a result of already handled events.

3.3 Graphics processor emulation

This part emulates the graphics processor's generation of visual graphics information. This work is to be regarded as non-urgent and is also not necessary for the correct execution of the c64 program. The urgent part of the work, i.e. the graphics processor's acquisition of data, is emulated by the bus handling.

The generation of graphic information takes place in a row, with 504 dots in each row. To make the generation as efficient as possible, each line is drawn in two steps. First, the graphics that lie as a background seen from the point of view of the moving graphic objects are drawn. Then, the graphics that originate from the moving graphic objects are drawn over the original background graphic. The drawing is done only in a table in memory of the points in the row, the visual drawing is handled by the program specific to the implementation.

The information for the drawing is retrieved from the stacks generated by the emulation of the bus management. Two stacks are used, one for the background graphics and one for the moving graphics. Since the information stored on the stacks is labeled with the exact time in relation to the graphics processor, it is just a matter of picking information from the stacks and the drawing proceeds.

3.4 Results from emulation

The core of the emulator is an object in C++ completely independent of computer systems. All communication between the real and virtual worlds represented in the core is done through well-specified functions and data formats.

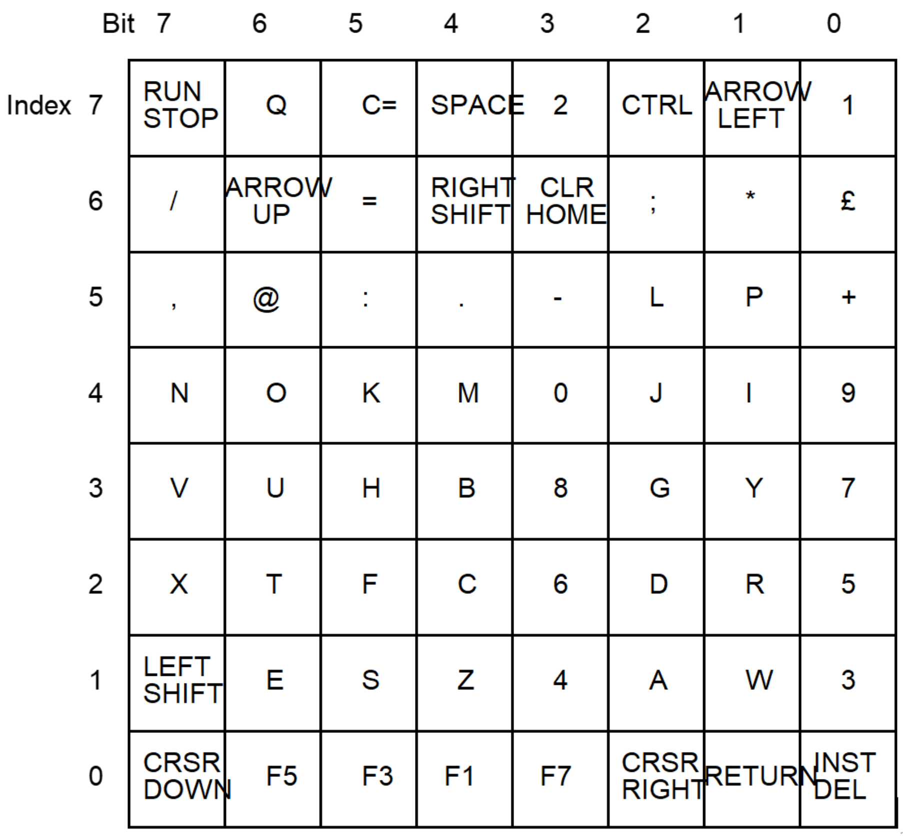
The start of the emulation is supported by total recovery and memory access capabilities.

The emulation itself is supported by functions for executing a certain number of machine cycles, reporting of peripherals such as keyboards, and more, as well as generating graphical information.

For the keyboard and peripherals, only changes are reported, as the emulator has its own internal representation of these. The keyboard is represented as a list of 8 bytes or, informally, an array of 8 by 8 bits, where each bit contains information about whether the key is pressed (0) or released (1). The pieces in the matrix have meaning according to the following sketch.

The generation of graphical information is done in a row, and the row is represented as a list of integers, with each integer corresponding to the color of a pixel.

Below is a block diagram of the core of the emulator and a possible implementation.



Index \ Bit	7	6	5	4	3	2	1	0
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

Figure 3.1: The internal representation of the Commodore 64 keyboard's status.

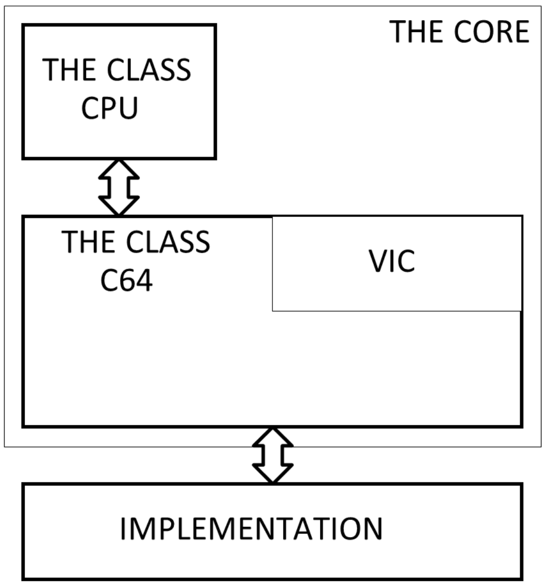


Figure 3.2: A block diagram of the core of the emulator in relation to implementation.

4 Implementation

The emulator has been implemented for PC as well as for UNIX/X.

A major problem during the implementation was difficulties in obtaining documentation of the computer systems. For PC, it could also mean incorrect information. This was solved by searching through the Internet, which took a lot of time.

For implementation, a number of features need to be constructed that are specific to the intended system. The functions referred to are as follows:

- **Drawing graphics on the monitor**, i.e. a function that visually conveys the graphic information generated by the emulator to the outside world.
- **Reading the keyboard**. This is in effect an emulator for the keyboard that emulates the existing input device as a keyboard for the C64.
- **Reading of peripherals**. This feature emulates joysticks and more through the use of the existing resources of the system.
- **Real-time clock**. For the synchronization of the emulator with the outside world, a clock with an accuracy of 20 ms is needed.
- **Initialization of the ROM and RAM**. Before the emulator can be executed, an operating system and possibly a program must be loaded into the emulator's representation of memory.

The main principle for all implementations of the emulator is broadly as follows:

1. Initialize the emulator kernel.
2. Load operating systems and programs, send these to the emulator kernel.
3. Let the emulator kernel execute a number of machine cycles corresponding to a screen drawing.
4. If any external device such as keyboard etc. has changed, this is reported to the emulator kernel.
5. Let the emulator kernel generate a line of graphical information and draw it on the monitor.
6. If less than 20 ms have passed since step 3 was performed, then perform step 5 again.
7. If the execution of the emulator is to continue, perform step 3 again.

4.1 Implementation for PC

The implementation for PC has been written in C++ and assembler. I have chosen to write the program for MS-DOS as the graphics handling in Windows is relatively very slow. One reason why some parts have been written in assembler is that there is very limited support for graphics, keyboards, and more in C++ for MS-DOS. In addition, it is very significant for speed to write certain hardware-related parts in assembler.

I have chosen to use the entire screen area for displaying the graphics. The visible image size of a C64 is 384 by 282 pixels. Regular VGA has a resolution of 320 by 200 pixels. However, the hardware for VGA can be reprogrammed so that the resolution for the C64 is obtained [4]. Due to compatibility issues between different VGA graphics cards and computer monitors, I have constructed 4 different graphics modes. They are 320 times 200, 368 times 240, 384 times 246 and 384 times 282 pixels.

The keyboard is implemented entirely in hardware. A key on the keyboard in the C64 corresponds directly to a key on the PC keyboard. Joysticks are also implemented as keys.

One problem that arose during the implementation was that MS-DOS can only handle 640 kbytes of memory in a useful way. In addition, compiled sections of program code or lists of data cannot be larger than 64 KB. This led to the kernel not being compilable. To solve these problems, I had to use a so-called DOS extender (DPMI) [5]. This program enables programs that use a linear memory model to be executed under DOS.

4.2 Implementation for UNIX/X

The implementation for UNIX/X has been written in C++. No assembler has had to be written as Unix and X-Windows provide sufficient functions.

To display the graphics, an X-Windows window with a size of 384 by 282 or 768 by 564 pixels is used. The latter size of window displays each pixel of the C64 as 2 by 2 pixels. The increased window size may be needed as a regular computer monitor for X-Windows commonly has a resolution of 1024 x 768 pixels, and then the graphics become blurry in the relatively small window size. To display colors, an X-Server with support for 256 colors is required. The X-Server can be quite slow as the graphics information is sent over a computer network. To solve this, I have constructed a so-called cache memory that represents the entire monitor. Only when new graphics information differs from the cache memory is it sent to the X-Server.

The keyboard is implemented entirely in hardware. A key on the keyboard in the C64 corresponds directly to a key on the workstation keyboard. Joysticks are also implemented as keys. Since some non-alphanumeric keys are not available on all keyboards, these keys may be implemented differently for different keyboards. For this purpose, there is a text file describing the configuration of the keyboard, this text file may be adapted to the current keyboard.

4.3 Results from implementation

I have achieved two implementations, one for PC with MS-DOS and one for UNIX with X-Windows. These implementations are relatively similar, both in functionality and in structure. About 90 percent of the source code for these implementations is common and these consist entirely of the so-called core of the emulator. This kernel is identical in both implementations as only additions to the kernel were needed.

For UNIX/X, the emulator has been compiled for two different computer architectures. These are SUN workstations with SPARC processor, as well as for computer architectures built on the DEC ALPHA processor.

5
Testing

To test the emulator, I have used a very large number of C64 programs. I have transferred these programs from the media that C64 uses as standard, cassette tapes. To transfer the programs, I have used specially designed hardware for the AMIGA computer system, which enables the connection of the special tape recorder used to the C64.

The programs are mainly games, but programs intended for program development and debugging on the C64, so-called machine code monitors, have also been tested. With the help of these, I have been able to test the emulation very thoroughly at the hardware level, as these machine code monitors can show pretty much all information and status about the hardware in the C64.

I have also used a commercial debugging program for UNIX called Purify.

The testing has led to several bugs in the emulator being fixed with additions and changes. The testing has been repeated until no errors have been detected.

A problem with the testing was that some errors that were discovered were more often due to errors in the documentation of C64 than pure programming errors, which led to more research with increased work effort as a result.

5.1 Results from testing

With the exception of audio, the following programs have demonstrated total accuracy:

- Programs that do not allow any access to other files or external devices except joystick. At least 99 percent of all programs for the C64 are of this type.

Other programs that require external devices have shown overall accuracy compared to a original C64 without these external drives.

To get the correct speed of implementation for PC, a processor of type 486DX2/66 is required. Faster processor means faster updating of the graphical information.

To get the correct speed of implementation for UNIX/X, a minimum of a processor of type SPARC 40 Mhz is required. Running on computers with a processor of type DEC ALPHA gives a very good speed, computationally as fast update as the original C64. However, the update of graphical information is limited by the network and the X-Server.

6 Conclusion

The Commodore C64 has a relatively complicated piece of hardware. This hardware is now documented in pretty much every respect in terms of functionality.

Emulation of the C64 is possible, in addition, it is possible to manufacture a kernel of the emulator that is system-independent. Such a kernel has been manufactured, written in the ANSI C++ programming language. The basic ideas of this kernel should be useful in the construction of emulators of other older microcomputer systems as these often have similar architecture.

Two implementations of an emulator for C64 have been made, one for PC and one for UNIX/X. Both are built on the manufactured emulator core. For the implementation on PC, MS-DOS is used. The implementation on UNIX/X has been compiled on two different computer architectures, SUN SPARC and DEC ALPHA. Compilation should be possible on all UNIX/X architectures, with only small changes to the Makefile.

Implementations on most other modern computer systems should be possible with relatively little effort.

The emulator for the C64 has with few restrictions shown very good accuracy.

Bibliography

[1] Commodore Business Machines (1982). Commodore 64 Programmer's Reference Guide, Howard W. Sams & Co.

[2] Mäkelä, M. et al. (1994). Documentation for NMOS 65xx/85xx Instruction Set, USENET

[3] Mäkelä, M. et al. (1994). The memory accesses of the MOS 6569 VIC-II, USENET

[4] Shaggy of The Yellow One (1994). Documentation Over the I/O Registers for Standard VGA Cards.

[5] The DPMI Committee (1991). DPMI 1.0 Programming API Specfication.

A

User Documentation

A.1 User documentation for the C64 emulator for PC

This section describes how to use the PC version of the C64 emulator.

System:

- Processor Intel 486 33 Mhz or faster
- 4 Mb RAM
- MS-DOS version 5.0 or newer
- VGA

To start the emulator without a preloaded C64 program, type C64 on the command line.

To start the emulator with a preloaded C64 program, type C64 and then filename on the command line. The files with C64 applications that are supported have file names with suffixes. PRG or . T64.

Once the emulator is running, the keyboard works like the keyboard on a real C64 with these modifications:

Key	Description
Pause	Exit the emulator
Left Ctrl	C64 Commodore
Escape	C64 Run/Stop
Delete	C64 Arrow Up
Insert	C64 lbs
Home	C64 Clr/Home
Right Alt	Joystick 1 Fire
Numeric %	Joystick 1 Up
Numeric 5	Joystick 1 Down
Numeric 7	Joystick 1 Left
Numeric 9	Joystick 1 Right
Right Ctrl	Joystick 2 Fire
Numeric 8	Joystick 2 Up
Numeric 2	Joystick 2 Down
Numeric 4	Joystick 2 Left
Numeric 6	Joystick 2 Right

To change the graphics mode, edit the emulator.cfg file and modify as follows:

ID	Resolution	VGA Memory	Frequency
1	384 x 282 pixels	"Chained"	50 Hz.
2	384 x 246 pixels	"Chained"	60 Hz.
3	368 x 240 pixels	"Chained"	60 Hz.
4	384 x 282 pixels	"Planar"	50 Hz.
5	384 x 246 pixels	"Planar"	60 Hz.
6	368 x 240 pixels	"Planar"	60 Hz.
7	320 x 200 pixels	"Chained"	70 Hz.

A.2 User documentation for the C64 emulator for UNIX/X

This section describes how to use the UNIX/X version of the C64 emulator.

System:

- X-Windows system with X-Server terminal capable of displaying 256 colors.
- Terminal with keyboard.

If the program is not compiled, that is, the absence of a file named c64 that is executable for the intended system, the program runs make. Changes may be required to the Makefile file if the paths do not match.

To start the emulator without a preloaded C64 program, type c64 on the command line.

To start the emulator with a preloaded C64 program, type c64 and then filename on the command line. The files with C64 applications that are supported have file names with suffixes. PRG or . T64.

More options that can be obtained by adding this to the command line are as follows:

Syntax	Description
-1	Small window (384 by 282 pixels)
-2	Large window (768 by 564 pixels)
-sharemem	Uses an extension from MIT for the graphics. Can only be used if the program is running on the same terminal that displays the graphics as it can speed up the graphics update significantly.
-cache	Uses an internal buffer for the graphics. This can significantly speed up the graphics update as only new graphics are sent to the X Server. In addition, less system resources are used, such as network capacity.
-tilt	Turns the window 90 degrees clockwise. Can be useful for applications for C64 that require the TV or monitor to be placed on the edge.

Here's a complete syntax for the command line:

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

When the emulator is running and the mouse cursor is over the window, the keyboard works like the keyboard on a real C64 with respect to all normal alpha numeric keys. For the rest for C64 special keys, the file keys are used. This file defines for each particular key corresponding to the keyboard code on the existing terminal. When switching to a terminal with a different type of keyboard, these codes may need to be changed. These codes are hexadecimal and have the following designations and meanings:

Key	Description
keyPlus	Plus (+)
keyMinus	Minus (-)
keyEqual	Equals (=)
keyPound	Pound (£)
keyUpArrow	Up Arrow (↑)
keyAsterisk	Asterisk (*)
keyAt	Alpha (@)
keyColon	Colon, Left Bracket (: , [ )
keySemiColon	Semicolon, Right Bracket (; ,])
keyDivide	Divide, Question mark (/ , ?)
keyLeftArrow	Left Arrow (←)
keyCommodore	Commodore (C=)
keyRunStop	Run, Stop (RUN/STOP)
keyCtrl	Control (CTRL)
keyRestore	Restore (RESTORE)
keyClrHome	Clear, Home (CLR/HOME)
keyInstDel	Insert, Delete (INST/DEL)

For joysticks, the numeric keyboard and the Meta keys are used as follows:

Key	Description
Left Meta	Joystick 1 Fire
Numeric %	Joystick 1 Up
Numeric 5	Joystick 1 Down
Numeric 7	Joystick 1 Left
Numeric 9	Joystick 1 Right
Right Meta	Joystick 2 Fire
Numeric 8	Joystick 2 Up
Numeric 2	Joystick 2 Down
Numeric 4	Joystick 2 Left
Numeric 6	Joystick 2 Right

B

System Documentation

First, the system-independent kernel is described.

From a point of view, the core consists of a class C64 implementation. This is divided into three main blocks, CPU, EMULATOR and VIC.

The block CPU is made up of a class of CPU and is relatively independent from the rest of the core. However, it must have access to the block EMULATOR with class C64. In addition, class C64 must have direct access to the CPU class as some features of the block EMULATOR can produce side effects in the CPU class. Definitions of the block CPU can be found in the `cpu.h` file, and the program part can be found in the `cpu.cpp` and `cpu_defs.h` files.

For external communications, the CPU class has the following features:

> `CPU(C64 *c64)` - Constructor for the CPU class, the `c64` argument is a pointer to main class C64.
>
> `unsigned int Run(unsigned int cyclesleft)` - Executes emulation of microprocessor 6510 until class C64 reports a stop. The `cyclesleft` argument describes the number of machine cycles that remain to be executed until an event is to be handled. After a reported stop, the number of machine cycles remaining until the next event is returned.
>
> `void Set6510Regs(UBYTE a,UBYTE x,UBYTE y,UBYTE sr,UBYTE sp,UWORD pc)` - Assigns the internal representation of the microprocessor's records according to the respective arguments.
>
> `void Get6510Regs(UBYTE *a,UBYTE *x,UBYTE *y,UBYTE *sr,UBYTE *sp,UWORD *pc)` - Assigns the value of the arguments to the respective registers in the representation of Microprocessor.

The `Run` function makes use of the following internal sub-functions:

> `unsigned int DecimalAdd(unsigned int alu,unsigned int a,unsigned int c)` - Perform addition with the BCD numbers in the arguments `alu` and `a` and with the carry flag in the argument `c`. The answer is returned and the flags are set in the global structure `decimalregs` of type `DecimalRegs`.
>
> `unsigned int DecimalSub(unsigned int alu,unsigned int a,unsigned int c)` - Perform subtraction with the BCD numbers in the arguments `alu` and `a` and with the carry flag in the argument `c`. The answer is returned and the flags are set in the global structure `decimalregs` of type `DecimalRegs`.

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

The block EMULATOR consists of a class C64 and is the foundation of the core. However, it must have access to the block CPU with the class CPU. Definitions of the block EMULATOR can be found in the `emulator.h` file, and the program part can be found in the `emulator.cpp` files. Note that the block VIC is also included in class C64.

For external communications, class C64 has the following functions:

`C64()` - Constructor for class C64. This function also constructs an object of the CPU class.

`void Reset()` - Initializes all parts of the emulator corresponding to a cold boot of C64.

`UWORD Run6510(UWORD cycles)` - Executes emulation of the C64 system in the number of machine cycles corresponding to the `cycles` argument. The function returns the machine cycle, in reference to the graphics processor, at which execution stops.

`void SetKey(const unsigned int key)` - Changes the internal representation of the keyboard. The `key` arguments describe which key has been changed. The information in the key is interpreted as follows. Bit 7 indicates whether the key was pressed (1) or released (0). Bits 6-4 are the column for the key. Bits 2-0 are the corresponding line for the key. Bit 3 indicates whether the key should be interpreted as if it was combined with shift (1) or not (0).

`void SetJoystick(const UBYTE joy,UBYTE dir)` - Changes the internal representation of joystick numbers corresponding to the `joy`. The direction of the joystick is set according to the argument `dir`. The information in the `dir` is interpreted as follows. Bit 4 indicates whether the fire button is pressed (1) or not (0). The direction positions are indicated correspondingly in bit 3 for the right, bit 2 for the left, bit 1 for downwards and bit 0 for up.

`void SetPaddles(const UBYTE nr,const UBYTE x,const UBYTE y)` - Changes the internal representation of paddles numbers corresponding to the argument `nr`. The position of the paddle is set according to arguments `x` and `y`.

`void SetRom(const UWORD addr,const UWORD len,UBYTE* buffer)` - Copies to the internal representation of the ROM memory the number of bytes of the corresponding argument `len` starting with the address according to the `addr` argument. The source of the copy is a list of bytes according to the buffer argument.

`void SetRam(const UWORD addr,const UWORD len,UBYTE* buffer)` - Copies to the internal representation of the RAM the number of bytes of the equivalent argument `len` starting with the address according to the `addr` argument. The source of the copy is a list of bytes according to the buffer argument.

The `Reset` function makes use of the following internal sub-functions:

> `void InitMMU()` - Constructs the internal representation of the memory manager. It consists of eight lists of bytes. Each position in the list indicates the memory type of the corresponding address in memory.

The following functions are internal subfunctions that are called by the CPU class:

> `unsigned int ReadIO(const unsigned int reg)` - Reads a byte from an I/O device from records according to the argument reg. The complete address is according to the variable `addr` in the CPU class. The response is returned as well as the number of machine cycles remaining until the next event is put into the variable `cyleft` in the CPU class.
>
> `unsigned int WriteIO(const unsigned int data)` - Writes a byte according to the data argument to an I/O unit to register according to the reg variable in the CPU class. The complete address is according to the variable `addr` in the CPU class. The function returns the number of machine cycles remaining until the next event. If the write has an effect that cannot be executed immediately, information about the write is saved on the stacks `gfxsavedata` or `spriteSaveData`, which are global lists of type `GfxEvent` and `SpriteEvent` respectively. The stack used depends on whether the effect relates to the moving graphic objects (`SpriteEvent`) or not (`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()` - Checks if the bus is available for reading. If it is not free or if an event is to be handled, this is done until the bus is free for reading. The function returns the number of machine cycles remaining until the next event.
>
> `unsigned int WannaWrite()` - Checks if the bus is available for writing. If it is not free or if an event is to be handled, this is done until the bus is free for writing. The function returns the number of machine cycles remaining until the next event.
>
> `void SetExceptionNextCycle(unsigned int)` - Sets outages to be reported as an event the next machine cycle. The argument is the number of machine cycles remaining until the next event.
>
> `unsigned int SortEvents(unsigned int)` - Sorts events by time when they are to be reported. Takes as an argument and returns the number of machine cycles

remaining until the next event.

The following functions are internal subfunctions called by the `WannaRead` and `WannaWrite` functions:

> `void CheckEvent()` - Handles events.
>
> `void SortEvents(void)` - Sorts events by time when they should be reported.
>
> `void SortSpriteDma(const unsigned int)` - Sorts the moving graphic objects by time when DMA access is to be made. Takes as an argument the current time measured in machine cycles referred to the graphics processor.
>
> `void SaveSpriteData(const unsigned int)` - Saves data about the graphic information of the moving graphic objects. Takes as an argument the current line during drawing on the screen referred to the graphics processor. The information is stored in a list of lists of type `ULONG` as a global variable named `spriteDmaData`.
>
> `void ReadGfxDmaData(const int)` - Saves data about graphical information regarding the type of characters and color to be displayed on the screen, according to what the graphics processor reads from memory. Takes as an argument the column referenced to the graphics processor in which DMA access has been initiated. The information is stored in a list of integers as a global variable named `charDmaData`.
>
> `void ReadGfxRowData()` - Saves graphical information data according to what the graphics processor reads from memory for each line on the screen. The information is stored in a list of lists of integers as a global variable named `gfxDmaData`.

The following functions are internal subfunctions that are called by the `Run6510` function:

> `void FixEvents(unsigned int&)` - Checks if the graphics processor has passed the time for drawing an entire screen page, and if so, adjusts all events and accordingly. As an argument, the function takes the present time measured in machine cycles referenced to the graphics processor.

The block VIC consists of a class C64 and handles the generation of graphical information. However, it must have access to the block EMULATOR with its definitions. Definitions of the block VIC can be found in the `emulator.h` file, and the program part can be found in the `vic.cpp` and `vic_defs.h` files.

For external communications, class C64 has the following functions:

> `unsigned int* UpdateNextRaster()` - Generates the next line of graphical information from the graphics processor. The row is returned as a list of 504 integers, where each integer is the color of the point on the row that corresponds to the position in the list. As basic information, the function uses the stacks in the global variables `gfxsavedata`, `spriteSaveData`, `charDmaData`, and `gfxDmaData`.

The following functions are internal subfunctions that are called by the `UpdateNextRaster` function:

> `unsigned int UpdateScreenEvent(unsigned int)` - Handles events specific to background graphics. Takes as an argument and returns the number of machine cycles remaining until the next event regarding background graphics.
>
> `unsigned int UpdateSpriteEvent(unsigned int spritecy,unsigned int mask)` - Handles events specific to moving graphic objects. Takes as an argument spritecy and returns the number of machine cycles remaining until the next moving graphic object event. The mask argument describes which moving graphic object is affected by the corresponding bit set to active (1) or inactive (0).

B.1 System Documentation PC

This describes the main implementation program for PCs with MS-DOS. Definitions and program part of the main program can be found in the main.cpp and fixbios.asm files.

The program mainly consists of the following functions:

> `void addbios()` - Starts an interrupt handler for the keyboard and an interrupt handler for a timer that is set to give interrupt signals at 50 Hz.
>
> `void rembios()` - Restores the original interrupt handlers that were active before the call to addbios.
>
> `int readtimer()` - Returns how many interruptions were given from the timer set to 50 Hz.
>
> `int readrawkey()` - Returns the keyboard code that has been reported from the keyboard hardware, returns zero if nothing has been reported.
>
> `void initgfx(int)` - Initiates the graphics card to the graphics mode specified by the argument.
>
> `void restoregfx()` - Reverts the graphics card to the graphics state that was active before the call to initgfx.
>
> `void drawgfx(unsigned int *gfx,unsigned int dest,unsigned int len,unsigned int mode)` - Draws graphics on the screen from a list of integers according to the gfx argument, containing the number of color dots according to the len argument, to the graphics card memory starting with the address according to the dest argument. If the mode argument is zero, the drawing uses a so-called linear mode. Otherwise, the mode argument is used as the value for subtraction per drawing phase in so-called planar mode.

B.2 UNIX/X System Documentation

This describes the main application for implementing UNIX with X-Windows. Definitions and program part of the main program can be found in the main cpp file:

`main()` - Parses the command line and loads all the necessary files needed to run the emulation. Initiates everything to be able to start the emulation. Then, the emulation is executed by calling the emulator kernel, loading from keyboards, and drawing on-screen graphics until the application is canceled.

The following functions are subfunctions that are called by main:

`void OpenC64Window()` - Opens a window in X-Windows for drawing graphics.

`void CloseC64Window()` - Close the window.

`void CreateImage()` - Creates a reprint of the screen for the C64 in memory.

`void DrawGraphics(unsigned int)` - Plots graphics in the window in the number of rows according to arguments.

`int ParseEnvironment(char * name,char * var,char ** envp)` - Interprets the environment variables according to the argument envp. Looks for the environment variable by name according to the name argument, and assigns the argument to the content of the environment variable.

`void ParseOptions(int argc,char ** argv)` - Interprets the command line. The argc argument specifies how many different texts are on the command line, and the argv argument is a list of these texts.