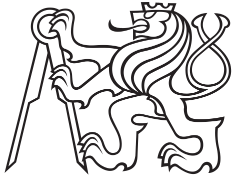
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Drivers and Libraries Enabling RTEMS OS GUI on Current PC Graphics Cards

Jan Doležal
Open Informatics, Computer Systems
dolezj21@fel.cvut.cz

January 2015
Supervisor: Ing. Pavel Píša, Ph.D.

Acknowledgement / Declaration

It is a pleasure to thank those who made the creation of this thesis possible. First of all, I want to express my gratitude to my supervisor Pavel Píša for his guidance and for supportive attitude. I also want to thank Gedare Bloom, Joel Sherrill and Rostislav Lisový for their truly valuable comments. Last but not the least my gratitude belongs to my family for their support in my studies and to my friends for refreshing distractions.

I hereby declare, that I wrote this thesis by myself and that I cited all used information sources in compliance with abiding ethical principles and methodical instructions which form the platform for the preparation of a university thesis.

In Prague, 6. 1. 2015

....

Abstrakt / Abstract

Hlavním cílem práce je implementace obecného ovladače kompatibilního s většinou grafických karet počítačů třídy PC (i386) pro operační systém RTEMS. Dosud byl pro tuto platformu dostupný ovladač pro standard VGA, který podporuje pouze nízká rozlišení a ovladač pro grafickou kartu Cirrus Logic, který vyžaduje zmíněný již zastaralý hardware, nicméně může být použit v rámci emulátoru QEMU. Nově implementovaný obecný ovladač překonává zmíněná omezení využitím rozhraní VESA BIOS Extensions. Toto rozhraní umožňuje volání kódu výrobce standardizovaným způsobem a je výrobci v grafických kartách ve velké míře podporováno. Seznam použitelných grafických režimů karty je dostupný voláním VBE rozhraní. Implementace ovladače volí pouze mezi módy s podporou přímého frame bufferu a získává adresu frame bufferu, aby mohla aplikace vykreslovat grafický obsah na výstupní zařízení. Ovladač byl testován s aplikacemi využívajícími Nano-X Window System, knihovnu FLTK nebo knihovnu SuiTk. Kód byl zahrnut do hlavní vývojové větve systému RTEMS.

Klíčová slova: RTEMS; VESA BIOS Extensions; ovladače grafické karty.

Překlad titulu: Ovladače a knihovny pro OS RTEMS podporující grafické karty současných PC počítačů

The main goal of this work is the implementation of generic driver compatible with most graphic cards on every PC compatible computer for RTEMS operating system. Two other PC graphic drivers are already included in RTEMS system, but the first one is the driver conforming to the VGA standard which supports only low resolutions and the second one is the driver for Cirrus Logic card that requires already obsolete hardware or can be used within QEMU emulator. Newly implemented generic driver overcomes limitations by usage of VESA BIOS Extensions interface. This interface allows to call vendor code in a standardized way and it is well supported across vendor cards. A list of graphic modes available on the graphic card can be obtained at the VBE interface. The driver implementation is designed to select solely modes with linear flat frame buffer and acquires the address of the frame buffer so that the application can draw its graphic content directly to the output device. The driver has been tested with applications utilizing Nano-X Window System, FLTK library or SuiTk library. The code of the generic driver was included in the RTEMS mainline.

Keywords: RTEMS; VESA BIOS Extensions; graphic card driver.

Contents

1 Introduction ... 1
1.1 Purpose of the research ... 1
1.2 Approach ... 1
2 Theoretical part ... 3
2.1 i386 architecture ... 3
2.1.1 Registers ... 3
2.1.2 Processor modes used in the work ... 5
2.1.3 Real mode ... 5
2.1.4 Protected mode ... 5
2.1.5 Addressing mechanisms ... 7
2.1.6 Jump and call operations ... 7
2.1.7 Interrupt mechanism ... 8
2.2 IBM PC compatible computer ... 9
2.2.1 BIOS ... 9
2.2.2 Video services ... 9
2.3 VESA BIOS Extension ... 9
2.3.1 Provided functions ... 10
2.3.2 VBE Interfaces ... 10
2.4 Selecting approach ... 11
2.4.1 16-bit protected mode interface ... 11
2.4.2 Real mode interface ... 11
2.4.3 Running real mode interface in VM86 mode ... 12
2.4.4 Utilizing i386 code emulator ... 12
2.5 RTEMS RTOS ... 12
2.5.1 License considerations ... 13
2.5.2 Source code organization ... 13
2.6 Toolchain ... 13
2.7 From source to install image ... 14
2.7.1 bootstrap ... 14
2.7.2 configure ... 14
2.7.3 RTEMS application ... 15
2.8 GCC ... 15
2.8.1 C extension - Extended Asm ... 15
2.8.2 Calling convention ... 16
3 Implementation ... 17
3.1 VBE and EDID header files ... 17
3.1.1 Packed attribute ... 18
3.2 Descriptor manipulation ... 18
3.2.1 Descriptor table modification ... 18
3.2.2 Functions dealing with descriptors ... 19
3.3 Real mode interrupt interface ... 19
3.3.1 Real mode related functions ... 20
3.3.2 Switching between real and protected mode ... 20
3.3.3 Calling the interface ... 21
3.3.4 Interface memory layout ... 21
3.4 Implemented frame buffer driver ... 22
3.4.1 VBE interface ... 22
3.4.2 Selecting graphics mode ... 22
3.4.3 Multiboot options ... 23
3.4.4 EDID ... 23
3.4.5 RTEMS frame buffer ... 23
4 Testing targets and debugging ... 25
4.1 Virtual hardware ... 25
4.1.1 GDB ... 25
4.2 Real hardware ... 26
5 Integration ... 27
5.1 Graphic demos ... 27
5.2 Code ... 28
6 Conclusion ... 29
6.1 Future work ... 29
References ... 30
A Specification ... 34
A.1 Specification in English ... 35
B Abbreviations ... 36

Tables / Figures

Tables

2.1. Purpose of i386 registers ... 3
2.2. i386 segmentation registers ... 4
2.3. Fields of descriptor table entry ... 6
2.4. VBE functions ... 10
2.5. RTEMS source directories ... 13
2.6. Configure options ... 15
4.1. Arguments used with QEMU ... 25

Figures

2.1. i386 register lengths ... 4
2.2. Addressing in real mode ... 5
2.3. Segment selector ... 6
2.4. Segment descriptor fields ... 6
2.5. PC memory layout ... 9
2.6. Steps from developer to user ... 14
3.1. Memory layout of real mode interrupt call ... 21
3.2. Packed pixel example ... 23
5.1. Nano-X demo malpha ... 27

Chapter 1
Introduction

The implementation of operating systems graphic drivers would be very challenging task if there should be a driver for each existing graphic card. Graphic card is a device that allows the computer to send visual data to the display as well as it provides ways to modify the visual data. There exist many graphic cards in the market and there are usually different ways to control each card. It would require high amount of time to implement drivers (software that enables to control graphic card) for all of them.

Graphic card and monitor vendors (associated in the VESA) proposed a solution by releasing a standard describing software interface for display devices named VESA BIOS Extensions. This standard defines the way to control display device — graphic card "without specific knowledge of the internal operation of the evolving target hardware" [1]. The standard has been widely adopted by hardware vendors. It would require enormous effort to support all graphic cards when developing small projects, but this interface enables us to implement generic driver allowing basic graphic operations in reasonable time.

1.1 Purpose of the research

RTEMS is a shortcut for the Real Time Executive for Multiprocessing Systems. The RTEMS is real time operating system. Real time system ensures that it responses to an event within certain time constraint. The RTEMS supports several processor architectures and several boards that use these architectures.

The i386 BSP of the RTEMS operating system already contains two frame buffer drivers. Frame buffer is memory designed to hold a screen frame. The memory contains consecutive chunks which represent the screen pixels. The first driver implements the standard VGA. However, there is a disadvantage in the implementation – the quite low resolution of 640x480 and used color depth of only 4 bits per pixel. The second frame buffer driver controls Cirrus CLGD 5446 PCI VGA graphic card and requires access of pixels by plane basis. This card is one of the cards simulated within QEMU PC System emulator – it currently supports resolutions up to 1280x1024 and bit depths up to 16 bits per pixel. The downside of this driver is that the specific peripheral is needed in order to run the software on real hardware. Emulators such as the QEMU are often used to test or to develop new software for various target machines.

In the response to above limitations new driver has been implemented to allow usage of wider range of graphic cards. The goal of this thesis is to describe its implementation.

1.2 Approach

The VBE Core Functions Standard document defines two interfaces. This standard is tightly bounded to x86 architecture. The processors which implement this architecture can operate in several modes. VBE functions can be accessed from real mode by using

interrupt mechanism to call the interface functions. The second option is to call 16-bit protected mode interface.

Since this implementation of the driver is designed for real time operating system RTEMS, the speed of graphics operations must be considered. Real time systems has to meet certain deadlines. As the vendor code is generated, it is crucial to maintain the control over the system for the period of the execution of the graphic operation.

Analysis of possible approaches is discussed in section 2.4.

Chapter 2
Theoretical part

In this chapter I am going to discuss the background knowledge and tools necessary for implementation of this particular driver. This includes: i386 processor that runs the code of the driver as well as the features of the board of the IBM PC compatible computer. Further, there are notes on the VBE standard which is the base for the implementation. This chapter also discusses the driver implementation considerations. Further, there is the description of tools used to build the RTEMS system and the description of the building process.

2.1 i386 architecture

The knowledge of the i386 architecture and its operating modes is required to describe driver implementation that uses low level CPU features and mode switching. This section introduces features of i386 architecture used in this work. In this section, there is used the document Intel 64 and IA-32 Architectures, Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C [2] as the main source of the information. I will use i386 throughout this work as a reference to all processors compatible with Intel 80386.

2.1.1 Registers

In comparison with other architectures i386 does not have many general purpose registers. This is the list of general purpose registers: `A`, `B`, `C`, `D`, `SI`, `DI`, `SP`, `BP` [2, Vol. 1 3–10]. i386 was originally designed as pure CISC architecture. It implies that each register has also its special purpose.

register	meaning	special purpose
A	accumulator	Accumulator for operands and result data
B	base	Pointer to data in the DS segment
C	counter	Counter to string and loop operations
D	data	I/O pointer
SI	source index	Pointer to data in the segment pointed to by the DS register; source pointer for string operations
DI	destination index	Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
SP	stack pointer	Stack pointer (in the SS segment)
BP	base pointer	Pointer to data on the stack (in the SS segment)

Table 2.1. Additional special purpose of i386 general purpose registers. [2, Vol. 1 3–11]

The accessibility of the parts of the general purpose registers in i386 architecture depends on the operand/address length. The veracity of this statement is going to be

demonstrated on the A register. If the A register is to be accessed as a 32-bit wide (double word) register, it is denoted as EAX. If it is to be accessed as a word, than lower word of EAX is used and it is called AX. Both bytes of AX are accessible, higher byte is called AH and lower byte is called AL. The same method is applicable to other registers as well – as it is shown in the picture.

31 – 16	15 – 8	7 – 0	*16-bit*	*32-bit*
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 2.1. i386 registers lengths. [2, Vol. 1 3–11]

Even though there exist also 64-bit version of registers which are labeled by the letter R – e.g. EAX is extended to RAX, it is of no use in solving this particular issue. Therefore, this thesis is not going to be dealing with the 64-bit processor modes which use this register length.

There are already mentioned 16-bit segment registers in the processor: ES, DS, CS, SS, FS, GS. The content of these registers influences the addressing of the processors. In principle these registers set a segment, which can be interpreted like a window in a memory where the data can be accessed. If there is memory not accessible from current segment/memory window, the segment register must be loaded by usage of a different value that grants the access to the desired part of the memory. The beginning of the segment/memory window is given by so called base. The described principle is called the memory segmentation.

Several segmentation registers are used as default if special operations are executed as described in Table 2.1.

register	meaning	purpose
SS	stack segment	along with SP determines current top of the stack
CS	code segment	segment from which instructions are read
DS	data segment	e.g. source data segment in string operations
ES	extra segment	e.g. destination data segment in string operations
FS	-	custom segment register
GS	-	custom segment register

Table 2.2. i386 segment registers. [2, Vol. 1 3–11]

The purpose of the EFLAGS register [2, Vol. 1 3–14] is to mark signs after the execution of the operation (instruction) and it is one of registers that control the state processor. Flags, such as zero or carry, allows conditional execution of the code.

Register EIP holds address of current instruction. Its content is automatically incremented by the length of the current instruction so that another instruction is executed after the first one finishes. The content can be also changed with jump and call instructions. Calls are basically used to execute a function.

The last register that is to be mentioned is `CR0` [2, Vol. 3A 2–14]. It is one of control registers. One of bits it contains is PE – protection enable. This bit allows to toggle protected mode.

2.1.2 Processor modes used in the work

There exist several operation modes in i386 architecture. RTEMS uses protected mode with default data length and the address length of 32-bits. The implemented driver requires running part of its code in real mode which is activated as the processor is started up.

2.1.3 Real mode

This mode is the first to be active as the processor is started up. This mode has been used for several decades for compatibility reasons. However, the real mode has limited memory access given by the evolution of the architecture. The accessible memory available after startup and under usual conditions is only first 1MB.

The segmentation mechanism allows to access 1MB of memory through 20-bit address bus using 16-bit registers. The pointer to memory is defined by the segment register and the offset register commonly written as *segment:offset*. To form the physical address, the segment register is shifted by four bits to the left and then the offset register is added. [2, Vol. 1 2–1]

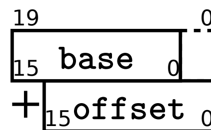


Figure 2.2. Addressing in real mode. The base is stored in the segmentation register and the offset in one of the general purpose registers.

2.1.4 Protected mode

The protected mode introduced several new features. The main feature of this mode is the availability of paging as well as the virtual memory [2, Vol. 1 3–7]. This allows the operating system to separate individual processes in a way that each process is able to operate with virtual memory of 4GB even though the size of real memory could be much lower. The size of the virtual memory is calculated as follows: the size of address bus which is 32-bits in basic protected mode – 2^{32} bytes equals 4GB. However this is not the feature which we are interested in throughout this work. In addition for the purpose of this thesis, I will consider paging to be turned off.

In the protected mode, segments are described with segment descriptors. Descriptors are stored in a GDT or in a LDT – Global/Local Descriptor Table. The position of GDT is determined by `GDTR` – GDT Register which has to be loaded with 6 bytes containing the base of GDT and its limit. LDT is described by one system descriptor in GDT and current LDT is stored in `LDTR` – LDT Register.

The segment register content meaning changed too. It is loaded with so called segment selector. The selector along with other general purpose register, forms protected mode pointer. The pointer is commonly written as *selector:offset*. The physical address is formed by addition of the offset part and the base part from the segment descriptor. The descriptor is indexed by the selector.

The segment selector bits 15–3 are the index to the descriptor table. Obviously, it is possible to express 2^{13} indexes, which would count 8192. This result is the maximum size of descriptor tables.

Figure 2.3. Segment selector. [2, Vol. 3A 3–10]

The bit 2 is the table indicator (TI) and when it contains 0, it indicates that GDT will be used. When the TI bit is 1 than LDT will be used.

The zero TI and the zero index to the descriptor table forms an invalid selector. In other words entry on index 0 in GDT cannot be used.

Bits 1 and 0 form the privilege level. The privileges are not going to be discussed in this document, I will just state that RTEMS system does not use the segment privilege level system. The privilege level value is always set to 0 – the most privileged level.

One descriptor table entry has 8 bytes and consists of fields shown in Figure 2.4.

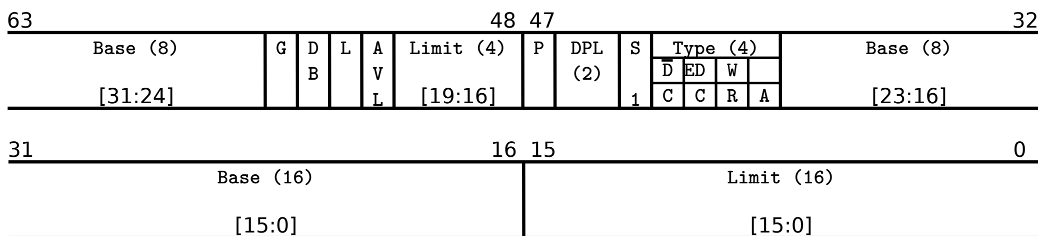


Figure 2.4. Fields of non-system GDT/LDT entry. [2, Vol. 3A 3–10]

field	meaning
Base	Base address of segment
Limit	Limit of the segment
G	Granularity (0 = limit unit is 1 byte; 1 = limit unit is 4kB)
D/B	Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
L	64-bit code segment
AVL	Available for use by system software
P	Segment present
DPL	Descriptor privilege level
S	Descriptor type (0 = system; 1 = code or data)
Type	Segment type

Table 2.3. Fields of descriptor table entry. [2, Vol. 3A 3–10]

Base address is a 32-bit integer specifying the beginning of the segment in memory. It serves as the physical memory address in case the paging has not been turned on or it does not remap addresses.

Limit is a 20-bit integer that indicates the last valid unit of a segment. If bit the G-bit is cleared then the limit value is in units of bytes. If the bit G is set then the limit value is in 4kB units. In this way, the whole addressable space – 4GB can be covered with the 20 bit value.

S specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or a data segment (S flag is set).

D/B if the segment is executable (contains code – see Type filed), the flag is called D and indicates default length of effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands

are assumed. If the flag is cleared, 16-bit addresses and 16-bit or 8-bit operands are assumed.

There exists instruction prefixes in the i386 architecture that change the default operand length or the address size if the instruction op-code starts with the prefix. The prefix 66H switches the default operand length and the prefix 67H switches the default address size.

For the stack segment the bit is called B (big) and specifies the size of stack pointer used for implicit stack operations. If the flag is set, the 32-bit stack pointer is used. When the flag is cleared, the 16-bit stack pointer is used.

P is the bit that indicates whether the segment is present in the memory. When the bit is cleared and the segment is accessed, then the processor raises an exception.

DPL determines the privilege of the segment addressed by this descriptor. In RTEMS, there is used only the highest privilege level of zero value.

Type bits depend on the bit S. This work is going to describe just types when S is set. The most significant bit of the type field – the bit 3 specifies whether the segment is executable/contains the code (the bit is set) or the segment contains data (the bit is cleared).

The code descriptor **type** field further contains the conforming bit (position 2), the readable bit (position 1) and the accessed bit (position 0). When the *Conforming* bit is set, then the subprogram in this segment will have the same privilege level as the calling segment, otherwise (the bit is 0) it will have this segment's privilege level. If the *Readable* bit is set to zero to make this segment unreadable, the only operation which could be made with the content is its execution. [3]

The *Accessed* bit is the same for both the code and the data descriptor. The processor sets this bit every time it accesses this descriptor in the descriptor table. [3]

The data descriptor **type** field further contains the expand down bit (position 2), the writable bit (position 1) and the accessed bit (position 0).

The *Expand down* bit indicates the direction of the segment expansion. Data segments contain basic data or stacks. If ED = 0 (data) the segment content will expand up to higher addresses from the base of the segment. If there is needed the achievement of the larger segment, the limit value has to be increased. If ED = 1 (stack) the segment content will expand down to lower addresses. The stack starts at the end of the segment and grows down to the limit (that is still counted from the base). If the segment is to be expanded, then the limit value is decreased. [3].

2.1.5 Addressing mechanisms

There are utilized several addressing mechanisms. The first one is the relative addressing. To address the range from -128 to 127 bytes, this addressing uses just a short offset from the current instruction. The second addressing mechanism utilizes the direct pointer contained within the instruction operation code. The third mechanism is the indirect addressing that reads the pointer value from the memory or the register. In this case the instruction operation code contains the place where the pointer itself can be found. [4]

2.1.6 Jump and call operations

This section refers to the instructions changing program execution flow. The instruction which follows the instruction currently executed is at pointer `CS:EIP` (or `CS:IP` in 16-bit context). There are two keywords used to reference whether the segment register is affected by the instruction changing place of program execution. *Near* indicates that

the instruction will operate only within the current `CS`. *Far* instruction will reload `CS` with the value.

Unconditional jump instructions change program execution flow without the option to return to the place of the jump. The jump can be either *near* or *far* and also relative (only in the *near* case), direct or indirect. The jump changes the content of `EIP` and `CS` (for the *far* jump).

The calling of instructions allows to temporarily change the place of the program execution and later return the program execution back. Basically these are instructions for the function execution. Again, it is possible to have *near* or *far* calls and to utilize the direct or the indirect addressing within the calls. When the call occurs the return values are stored to the stack. For the *near* call only instruction pointer register is stored to the stack and for the *far* call there are stored the `CS` register value and the instruction pointer value to the stack. In the end of its execution, the called code executes the `ret` instruction either in *near* or *far* version which returns the execution back to the instruction following the call instruction by the usage of the stack values.

Both the jump and the call have direct and indirect addressing versions. Direct addressing instructions contain the offset value in the instruction if the instruction is *near*. They contain the `CS` segment register value with the offset, if the instruction is *far*. Indirect addressing instructions operate with same pointer values for the *near* and the *far* instruction, but the pointer values are stored in the memory or in the register. The indirect addressing instruction contains the location (memory position or register) of the pointer.

2.1.7 Interrupt mechanism

The interrupt is similar to the *far* call, however the interrupt can occur asynchronously to the program execution. This is useful for handling events that occurred in outer world since processor is immediately informed about them by the means of sensors which activate interrupt mechanism. Interrupts are also raised by timers. The operating system can set a timer so that it starts the interrupt service routine periodically. It checks the running task and the OS can eventually perform operations such as the context switch. There is also the possibility to raise the interrupt in a software by the usage of the instruction `int`.

The interrupt is invoked with its number. The interrupt number is the index to the interrupt vector table. An entry of the interrupt vector table contains the pointer (segment register value and offset) to function that handles the interrupt. This function is usually called the interrupt service routine – ISR. When an interrupt is raised there are flags, the `CS` and instruction pointer registers stored to the stack. As all necessary steps for handling the invoked interrupt were made, the instruction `iret` is used to get back from interrupt service routine. It restores instruction pointer, the `CS` and flags registers from the stack.

The interrupt table base (location address) and the limit (last valid byte) is loaded to the `IDTR` (interrupt descriptor table register). The IDT can be relocated by loading the `IDTR` with a different base. In the real mode each entry has 4 bytes consisting of pointers *segment:offset* to ISRs. In the protected mode entries are 8 bytes long. In the protected mode the IDT entry may contain the pointer *selector:offset* to the ISR.

2.2 IBM PC compatible computer

IBM PC compatible computers are widely spread machines that operate with the i386 compatible processor. Several features specific to the IBM PC compatible computer are going to be introduced below.

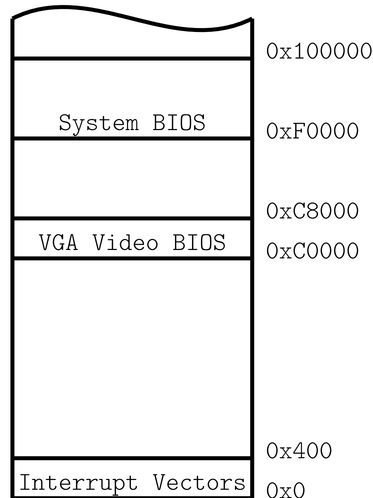


Figure 2.5. Places in PC physical memory space after startup. [5–6]

2.2.1 BIOS

The Basic Input Output System is a kind of firmware. It serves for initialization of hardware after every start of the computer. Another function of the BIOS is to serve as a compatibility layer between the underlying hardware and the overlying software – hardware abstraction layer. [7] The BIOS provides services to be used by the software and they are available through the internal interrupt mechanism – the interrupt raised by the software. [8] It works in a following manner. When a service is to be used, it stores its parameters to the registers. Subsequently, the ISR is called by the usage of the i386 instruction `int X`, where X stands for the interrupt number under which the service is installed. These interrupts are installed during boot up phase by the BIOS itself. The majority of the services code is just **compatible** and can be called **in the real mode**. [9]

2.2.2 Video services

Among others, there is the service for graphical operations installed under the interrupt number 0x10. As this interrupt is invoked, the code in the Video BIOS is executed.

2.3 VESA BIOS Extension

The VBE is a standard which "is intended to simplify and encourage the development of applications that wish to use graphics, video and audio devices without specific knowledge of the internal operation of the evolving target hardware". [1] Manufacturers of video cards can provide this interface by implementing its functionalities into the System Video BIOS. When there is a reference to the VBE standard in this document, it refers to the version 3.0 of this standard.

2.3.1 Provided functions

The VBE defines several functions to allow unified basic management of graphic cards. To inform the adapter code that VESA Extension functions are requested, the 0x4F value is 0x4F passed in the `AH` register. The function number is passed in the `AL` register. The sub-function number, if required, is passed in the `BL` register.

function name	retrieved/delivered information
Return VBE Controller Information	VBE version, hw capabilities, mode list, strings describing product and vendor
Return VBE Mode Information	resolution, bits per pixel, physical address for flat memory frame buffer
Set VBE Mode	mode number, frame buffer model
Read EDID [10]	selected EDID block from display

Table 2.4. Selected VBE functions. [1]

The graphical mode describes how the graphic adapter will set the output to the display if the mode is selected. Modes have parameters such as resolution or bits per pixel and a way how to pass viewable data to be shown on display – in other words the type of frame buffer and its parameters.

The frame buffer is a mechanism to "draw on the display screen". The frame buffer is located in the physical memory and it is accessed by its mapping to the physical memory space of the computer in this case. The frame buffer memory represents consecutive pixels. The bits per pixel field in mode information defines the space covered by one pixel in the memory. The linear or the flat frame buffer indicates that there are all screen pixels available at one time. There exist another approach that lowers the amount of address range required to map the frame buffer. It is called the banked frame buffer and there are several banks of pixels mapped to one place in the memory. Each bank represents a part of the display. As there is only one bank active at one moment, it implies that is possible to draw only to a certain part of the display. There is a mechanism called the bank switching that selects which bank will be available in the mapped memory – which part of the display can be redrawn.

2.3.2 VBE Interfaces

The standard itself defines several interfaces to access the subset of functions. It is up to the manufacturer if these interfaces are supported.

The real mode interface has been used since the first version of the VBE. If the manufacturer supports the VBE then functions of this interface will be available unless it is a "stub" implementation [1]. This interface extends video services provided under the real mode interrupt service number 0x10.

The subsequent interface was introduced with the VBE in the version 2.0 and it is called the "VBE 2.0 Protected Mode Interface" [1]. It is the 32-bit protected mode interface but it does not include functions for obtaining graphic mode info or for setting the mode. It is meant to be used along with the real mode interface and functions included in this interface are meant to speed up graphical operations such as the bank switching. The manufacturer is not obliged to implement this interface. The probability of the availability of this interface also decreased when the protected mode interface, described in the next paragraph, was introduced.

Another interface introduced with the VBE 3.0 is once again a protected mode interface. This time, it is build as 16-bit code. The video BIOS code is intended to be a 'dual-mode' which includes the "new protected mode interface entry point" which is to be called directly from the protected mode. BIOS vendors are not obliged to implement this interface. [1]

There is one more interface I would like to mention and that is the VBE/AF – VESA BIOS Extension Accelerator Functions [11]. Essentially, this standard defines the software layer implemented in the operating system. This layer interconnects the software, which uses the accelerated graphic functions, with the underlying hardware. This VBE/AF driver needs to support many graphic cards to be able to provide generic functions to the software that wants to use the driver. The implementation of such a driver would require too much time. The driver would have to support many graphic cards that are available in the market in order to make this solution portable. There were projects trying to implement this type of driver e.g. FreeBE/AF[1].

2.4 Selecting approach

This section discusses possible ways of the implementation of the VBE frame buffer driver. The base for this section is the section 2.3.2 in which are summarized standards–defined interfaces which are available. Below, there are going to be discussed features of the architecture and the RTEMS system features. The specific problems of the interfaces which are to be put into consideration are mentioned in this section, too.

2.4.1 16-bit protected mode interface

This optional interface can be called directly from the protected mode. The RTEMS runs in the 32-bit protected mode so the call requires to switch to the code selector that points to the descriptor with the 16-bit default operation size, so that the VBE code runs as the 16-bit code. The next step is to find the protected mode info block structure within the first 32 kB of the graphical BIOS. In this structure, there is the protected mode entry point that can be called.

This approach was implemented at first, but it turned out that the interface is not well supported by manufacturers or their implementations are broken. The VBE code would require to be patched [12], which gets us close to card/manufacturer specific drivers.

2.4.2 Real mode interface

This interface requires the processor to be in the real mode. The call of the interface functions may be executed before the RTEMS executive scheduler is started or the scheduler has to be stopped and all interrupts has to be disabled to switch safely into the real mode. The first approach has been selected. The main reason why not to call the real mode interface while there is the RTEMS executive running and interrupts are enabled is that the interrupt handlers code would not be interpreted correctly, because it is compiled as the 32-bit code for the RTEMS, while the real mode runs the 16-bit code.

If this interface is to be used, then all operations related must be called upon the RTEMS executive start while there is no critical code running. It follows that the graphic mode will be set on the system startup and could not be changed later. That

[1] `http://www.shawnhargreaves.com/freebe/index.html`

should not be a problem, because real time systems would rarely need to change the mode while running anyway. The graphic mode can be set either before the switch to the protected mode or utilizing switch from the real to the protected mode and back again. The RTEMS allows to be compiled as a multiboot image and the loader that runs the image might switch to the protected mode before the RTEMS code starts its code execution. Therefore switches between the two processor modes are the better way to go.

After the fail of the utilizing the 16-bit protected mode interface discussed in previous section I used this approach which was a safe bet.

2.4.3 Running real mode interface in VM86 mode

i386 processors contain the mode for the virtualization of the 8086 processor. That allows to run the real mode code while the processor is in the protected mode. Using this method to call VBE functions would require to rewrite the context switch function of the RTEMS operating system to support the switching among the VM86 task and protected mode tasks. That would introduce undesirable slowdown of the context switch.

This could also run on the executive startup but that introduces needless complexity in a comparison with switching to the real mode and back to the protected mode.

2.4.4 Utilizing i386 code emulator

This solution requires implementing or porting the x86 code interpreter. There is no overhead to the context switch as there is in previous cases. Further this would allow to monitor emulated code actions. This option seems to me as quite complex even though it would be probably the best way of those mentioned to implement this driver. The implemented instruction emulator exists in the Seoul Virtual Machine Monitor project[1].

2.5 RTEMS RTOS

The RTEMS[2] is the Real Time Operating System. Real time *systems* in general “must receive and respond to a set of external stimuli within rigid and critical time constraints” [13]. The usage of real time *systems* is truly wide, they are used for example in the car industry, avionics, manufacturing control mechanisms or the hospital equipment. Real time *operating systems* have equivalent purpose to the general-purpose operating systems and that is to provide and manage available resources in the whole system implying an easier software development. Further, real time systems contain schedulers that schedule running tasks under given time constraints if that is possible.

The name RTEMS is the abbreviation for Real Time Executive for Multiprocessing Systems. System has been ported to many processor families and boards build upon a specific processor. [14]

The RTEMS is available “in the form of source code” [15] That allows the high reusability of this software. Tools for building binaries from the RTEMS source code are described in the section 2.6. Further there exists distributed and prebuilt packages.

[1] `https://github.com/TUD-OS/seoul/tree/master/executor`
[2] `http://www.rtems.org`

2.5.1 License considerations

The RTEMS is developed under the various free and open source licenses. Most of the source code is covered by the Primary RTEMS License. "The common characteristic of all of these licenses is that they allow linking with no requirements placed on the end user application."[1] The RTEMS project rejects any code that uses the GPL or the LGPL licenses, because the RTEMS License is not compatible with mentioned licenses.

Although the VESA driver implementation already exists for example in the Linux operating system, non of its parts could be used due to the license issues.

2.5.2 Source code organization

The RTEMS source code is divided amongst subdirectories in a way where architecture and target board specific code is isolated.

The non-portable source code tree contains separate code for the processors as well as for the board support packages. The board support package consists of code specific to particular board allowing the system to run on the board and enabling various board features.

directory	purpose
cpukit/score/cpu/*CPU*	CPU dependent files
c/src/lib/libcpu/*CPU*	CPU dependent support files
c/src/lib/libbsp/*CPU*/*shared*	Files shared over all boards
c/src/lib/libbsp/*CPU*/*BSP*	Board dependent files

Table 2.5. RTEMS source directories influenced by this work. [16]

2.6 Toolchain

The RTEMS is built using the cross development environment. The cross development means that "software development activities are typically performed on one computer system, the **build-host** system, while the result of the development effort is a software system that executes on the **target** platform". [13] The effect of cross development is that one powerful build-host system, powerful in the sense of fast development, can create software for many low-cost **target** devices.

"A key component of the RTEMS development environment is the GNU family of free tools." [15] GNU tools used in the RTEMS development includes Autotools, the GCC, Binutils or the GDB.

The generic C standard library used in the RTEMS is the Newlib that is intended to be used on embedded targets[2].

Quite recent version of Autotools is required to build the RTEMS. I installed the most recent versions of Autoconf and Automake from sources.

GCC and Binutils toolchain configured and build for the RTEMS is necessary. The GCC stands for the GNU Compiler Collection and it contains the compiler used to transform RTEMS sources to an executable program. "The GNU Binutils are a collection of binary tools. The main ones are **ld** – the GNU linker and **as** – the GNU assembler" [17]. It is possible to build these from the source as described in Getting Started with RTEMS [13]. I used packages prepared by my supervisor[3].

[1] https://www.rtems.org/license
[2] http://www.sourceware.org/newlib/
[3] http://rtime.felk.cvut.cz/debian/pool/

2.7 From source to install image

The Autotools system is used to generate scripts for the configuration and to generate files necessary for building with a minimal effort. The source code files written by the programmer need to be described for Autotools so that Autotools knows how to approach these sources. The programmer has to describe the sources in Autotools specific files.

The RTEMS source root contains the *bootstrap* script that executes certain Autotools steps necessary.

There are certain steps performed by the user of by the developer for testing the software after the running of *bootstrap* script. The `configure` script enables to set options that should be used during the build phase. The building itself is done with the `make` command which reads `Makefile` created by the `configure` script. The tools and linkable files created by `make` can be installed to a place selected in the configure phase by executing command `make install`.

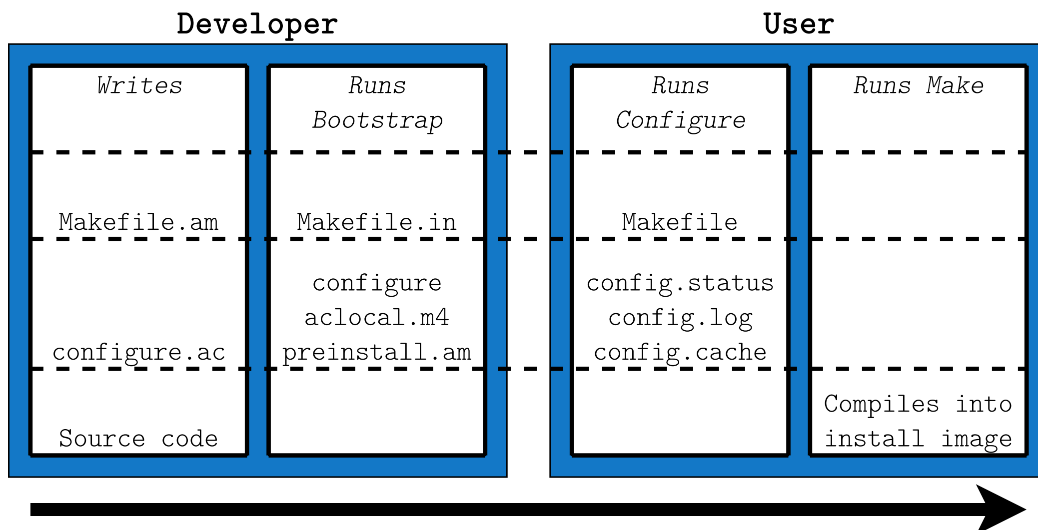


Figure 2.6. Steps from developer to user. [16]

2.7.1 bootstrap

The RTEMS *bootstrap* script utilizes GNU Autotools namely the GNU `Autoconf` and the GNU `Automake`.

The programmer writes the *configure.ac* which is an input file for the `autoconf` that generates the `configure` shell script. The *configure.ac* files consist of definitions of variables influencing what is build and how it is build. The user can modify or set these variables when invoking the `configure` script.

The programmer also writes the *Makefile.am* which is an input file for the `automake` which generates the file *Makefile.in*. The *Makefile.am* describes source and header files to be build incorporating `configure` conditions on which these files are included.

The *configure.ac* and the *Makefile.am* are found all around the source tree describing files in a particular subdirectory.

2.7.2 configure

The configure script provides ways to determine what will be included in the build following configuration. There are several options that can be used here. I used

the configure script with the following options: `./configure --target=i386-rtems4.11 --prefix=/opt/rtems4.11 --build=x86_64-pc-linux-gnu --enable-rtems-inlines --disable-multiprocessing --enable-cxx --enable-rdbg --enable-maintainer-mode --enable-networking --enable-posix --enable-itron --disable-ada --disable-expada --disable-multilib --disable-docs --enable-rtemsbsp=pc686`

The `configure` creates the Makefile s using the options provided and using files generated by Autotools.

option	purpose
--target=⟨*TARGET*⟩	⟨*TARGET*⟩ is of the form ⟨*CPU*⟩-rtems4.11
--prefix=⟨*INSTALL_POINT*⟩	installation point for the tools
--build=⟨*BUILD_HOST*⟩	host on which the software is being built
--enable-rtemsbsp=⟨*BSP*⟩	selects board support package to be built

Table 2.6. Configure options. [13] Other configure options may be obtained by executing `configure --help` command

2.7.3 RTEMS application

The RTEMS operating system binaries are in the install point selected in the configure phase. When an RTEMS application is written, its sources are first compiled to the relocatable object file [name].o. The application object file is then linked together with RTEMS prebuilt binaries and that results in a statically linked [name].exe image file. For the pc386 BSP there are removed .comment and .note sections and any unneeded sections and this is stored in a [name].bin image file. This [name].bin file is converted to a bootable image file [name].ralf that includes the RTEMS switch from the real to the protected mode. The tool for creating the bootable image is the bin2boot and its source code is a part of the pc386 BSP.

2.8 GCC

The following section is going to describe some of the information specific for the GCC compiler used with this project.

2.8.1 C extension - Extended Asm

The GCC allows to use a special construction to write an architecture specific assembly code mixing it with the C code. This feature is also known as the inline assembler. In the C code it is introduced as an `asm(code : output operands : input operands : clobbered registers);` the function, where the first argument is a string representing code in the machine specific assembly language including possible instruction separation by the `\n` newline placeholder. Further there are optional parts that are separated by colons. These comma–separated parts are intended to inform the compiler what is happening in the inline assembler. That makes the compiler's job easier when introducing possible optimizations. [18]

If the compiler stores something to registers or on the stack it might expect these to still be there after operations in the inline assembly were executed. However, while in the inline assembly we often need to use the registers. It could be the temporary registers or the specific registers used by the instructions. Here comes the clobbered

list that informs compiler about registers (or eventually memory) that are being used so it knows it needs to preserve these somewhere. [18]

Input and output registers allow programmer to pass variables or other data to and from the inline assembly. There are constraints on operands passed defined to make the work of the compiler easier. Constraints include the different types of memory, register, immediate or memory address operands. Each architecture may also have special constraints e.g. some of i386 register constraints are `a, b, c, d` for corresponding registers. [19]

2.8.2 Calling convention

The code compiled from the assembly and from the C language source files can call functions from the code which originates in the other source type. Assembly code has to adhere to the C application binary interface (ABI) calling convention. The convention describes among other how parameters are passed, the way the return value is passed, what registers are preserved and whether the stack is handled by a caller or a callee upon return. It is usual to pass parameters in the registers or on the stack. There does not exist only one calling convention, but there are more of them defined.

On the i386 in the 32-bit mode the GCC uses a calling convention that passes parameters in a reversed order on the stack, the return value is passed in the `A` register. Registers `A`, `C` and `D` are used as scratch registers so these do not need to be preserved. All other registers need to be preserved by the callee. The caller removes arguments from the stack after the callee finishes and returns. The function call is done using the `call` instruction and return back using the instruction `ret`. [20]

Chapter 3
Implementation

There were several steps necessary in implementing the driver utilizing the VBE real mode interface (see section 2.4.2). One step was preparing a header file describing the VBE data structures and macros. The RTEMS had to be extended to provide functions for the manipulation with global descriptor table entries. The implemented VESA based driver requires setting up some of the global descriptors. Further the driver needs to call the real mode interrupt, which lead to the implementation of the real mode interrupt interface. With all previous steps achieved, the frame buffer driver itself could be implemented.

In every section there are listed the most important files related to the section.

3.1 VBE and EDID header files

c/src/lib/libbsp/i386/pc386/include/vbe3.h
c/src/lib/libbsp/i386/pc386/include/edid.h

The necessity for creating own VBE header file arose, because I had not found such a header file under license compatible with the RTEMS anywhere.

For early version of the VBE header file I wrote a simple python script[1]. The script used the pdf2txt[2] tool to convert the VBE specification to a parsable text. There are mode list, function numbers and structures retrieved from the text file. The resultant file was further completed and adapted manually.

I started writing manually only necessary parts of the EDID ending up with structures in versions 1.4 and 2.0 of EDID definitions written. Nevertheless there was need to keep definitions endian independent and the first implementation did not satisfy that condition. Also “the EDID data structure 2.0 defined in the EDID Version 3 Standard has not been widely adopted, although the standard is still considered valid” [21] and “new designs are strongly urged to use only the new data structure 1.4” [21]. These things lead me to drop the support for the EDID version 2.0 and rewrite only the EDID structure version 1.4 to be an endian independent.

The endian independence of the EDID header file is a factor that makes its definitions multi–platform. There is a real possibility that it will be used on the different architecture. The VBE header file on the other side is based on the standard that is very dependent on the IBM PC compatible computer so there was no need to make it platform independent.

[1] https://github.com/dolezaljan/vbe_headers_tool
[2] http://www.unixuser.org/~euske/python/pdfminer/

3.1.1 Packed attribute

There are many structures defined with a compiler attribute *packed*. “This attribute specifies that each member of the structure is placed to minimize the memory required.” [22] The compiler could incline to align members at a certain boundary e.g. every eighth byte which might optimize the access to the members of a structure. The extra space is undesirable because members of underlying structures from used standards are naturally defined with the exact spacing.

3.2 Descriptor manipulation

To support the global descriptor table entries manipulation in the RTEMS, there was added an option to change the size of the GDT and there were written functions enabling to populate the GDT.

3.2.1 Descriptor table modification

c/src/lib/libbsp/i386/pc386/include/bsp.h
c/src/lib/libbsp/i386/pc386/include/tblsizes.h
c/src/lib/libbsp/i386/pc386/startup/ldsegs.S

The size of the GDT was fixed in an assembly (ldsegs.S) even though the bsp C header contained a macro `GDT_SIZE` defining the size of the GDT. When an assembly file has an extension `.S`, it is preprocessed using the `as` internal preprocessor with GNU C compiler driver. It means that it might be possible to include the bsp.h C header file and utilize the `GDT_SIZE`. Doing that the `as` utility threw an error, because the header file contains constructions such as function prototypes, that stays in the code after the preprocessing and that cannot be processed by the `as`. Nevertheless for defines this is not an issue. The first implementation introduced guards in the bsp.h header file checking whether the header file is processed within the assembly context. When it is in the assembly context, guards ensured that all parts besides those desired by assembly, such as the size macro, were ignored. The code using these guards became a little bit hard to read. So then I moved the GDT_SIZE macro to a separate file and included it from both the bsp header file and the assembly file.

The ldsegs.S contains a table of three predefined segment descriptors (including a null segment) and then it continues with the allocation of the space for the rest of the descriptors up to the GDT_SIZE. The allocation is done using the pseudo instruction `.rept` *count*, that repeats “the sequence of lines between the `.rept` directive and the next `.endr` directive *count* times” [23]. The repeated lines define a descriptor element. One descriptor element has 8 bytes and I set its bytes to 0 and consider it an empty descriptor. Using the GDT_SIZE there is also updated the limit of the structure that is loaded to the `GDTR` later.

Similarly the construction of an interrupt descriptor table was updated using the IDT_SIZE.

Further there was introduced the macro `NUM_APP_DRV_GDT_DESCRIPTORS` included from the file bspopts.h that is generated during the configure phase. It might be either set as an shell environment variable either passed as an argument variable with the number of descriptors required behind an equal sign.

3.2.2 Functions dealing with descriptors

c/src/lib/libbsp/i386/shared/irq/idt.c
c/src/lib/libcpu/i386/cpu.h

Several functions for manipulating descriptors of the GDT were implemented. The most important is the function for installing the GDT entry – `uint32_t i386_raw_gdt_entry(uint16_t segment_selector_index, segment_descriptors* sd)` where the index selects the entry of the GDT that is filled with values of `sd`. The function checks whether the provided index is within the limit contained in the `GDTR` and if it is different from 0, which is an invalid index in the GDT. It first sets the present bit to 0 and when all other fields are filled to the GDT then the value of present bit that was passed is filled. Although the bit is intended to different purposes, here this should prevent an application from using descriptors that are not fully prepared. If an application tries to use the descriptor that is not ready, its present bit contains 0, the processor generates a segment-not-present exception". [2, Vol. 3A 3–11] After the descriptor was filled the segment registers are reloaded with the selectors they contain so possible changes take effect.

Functions `i386_fill_segment_desc_base` and `i386_fill_segment_desc_limit` were implemented as supporting functions to fill individual parts of the base and the limit. Complementary functions to those mentioned above are the `i386_base_gdt_entry` and the `i386_limit_gdt_entry` that return the base eventually limit of the descriptor when pointer to the descriptor is passed into the functions.

Another function `i386_next_empty_gdt_entry` returns the index of another unused descriptor until their exhaustion. In the case of the exhaustion 0 is returned which is an invalid index in the GDT.

The function `i386_cpy_gdt_entry` copies the entry of the GDT on the passed index to the structure which pointer is passed to the function as well.

The pointer to the descriptor of the GDT on the given index is returned by the function `i386_get_gdt_entry`.

The existing function `i386_set_gdt_entry` is reimplemented under the name `i386_raw_gdt_entry`. The original function remains for the backward compatibility. Some of the code of the original function was reused in the `i386_raw_gdt_entry`.

3.3 Real mode interrupt interface

c/src/lib/libbsp/i386/shared/realmode_int/realmode_int.c
c/src/lib/libbsp/i386/shared/realmode_int/realmode_int.h

The interface enables the calling of the given interrupt number with the content of the registers defined. The interface entry point is called from the protected mode it switches to the real mode, executes interrupt and switches back to the protected mode. The interface also provides the buffer accessible from the real mode.
To comfortably work with the real mode pointers there was a need to implement functions converting the pointers to the physical address and vice versa.

3.3.1 Real mode related functions

cpukit/score/cpu/i386/cpu_asm.S
cpukit/score/cpu/i386/rtems/score/i386.h

Converting the real mode pointer to the physical address means to shift the segment part of the pointer 4 bits left (it is the same as multiplying by 16) and then to add offset part of the pointer to the shifted segment.

Converting the physical address to the real mode pointer is a little bit trickier. The highest physical address accessible using the real mode pointer is 0x10FFEF. Indeed if we have the pointer 0xFFFF:0xFFFF then shifted segment 0xFFFF0 plus offset 0xFFFF is 0x10FFEF. So if the function gets higher physical address to be converted it returns an error value. I also implemented the function to compute the highest segment part possible.

3.3.2 Switching between real and protected mode

The problem of the switch from the protected mode to the real mode and back requires couple steps to be addressed [2, Vol. 3A 9–12][24].

The code switching to the real mode suppose it switches from the 32-bit protected mode. The code also prepares to switch back into the 32-bit protected mode. It has to do the following:

- Turn off the paging. The code is expected to run only on the startup of the RTEMS executive when the paging is not yet active, therefore the paging is only tested and if it is on, an error value is returned.
- Prepare the entry pointer into the real mode.
- Prepare the return pointer into the protected mode.
- Disable interrupts.
- Back up selectors from segment registers and the `ESP` register that along with the `SS` describes the stack.
- Back up the current interrupt descriptor table register – the `IDTR` and set the `IDTR` to use the table located in the first MB of the memory so that it is accessible in the real mode. I actually set it to correspond with the table that is set up by the BIOS after startup.
- Load `CS` register with the real mode alike code descriptor. That means I needed to create a descriptor that has the 16-bit default operation size, byte granular limit with a value of 0xFFFF, non-conforming and readable. The load of the `CS` is performed by the far jump instruction that loads the `CS` register and the `EIP`.
- Load other segment registers with the real mode alike data descriptor that uses the 16-bit operand size, has the byte granular limit with the value of 0xFFFF, expands up and is writable.
- Disable the protected mode by setting the bit PE in the `CR0` register to zero.
- Load the `CS` register with the real mode base. Again using the far jump instruction.
- Load other segment registers with appropriate values of the real mode base.
- Establish the real mode stack.

Now we are in the real mode. To transfer back to the protected mode state we left earlier we restore backed up values and we perform other necessary steps.

- Set the PE bit in the `CR0` to enable the protected mode.
- Execute the far jump to previously saved return pointer incorporating previous 32-bit segment descriptor.
- Load other segment registers with previous 32-bit segment descriptors.
- Restore the protected mode stack.
- Restore the interrupt descriptor table.
- Enable interrupts if they were enabled prior to the call of the interface.

It is obvious that when transferring back to the protected mode there is not necessary to load the segment selectors pointing to the 16-bit segment descriptors again.

3.3.3 Calling the interface

Originally the interface had one parameter and that was the pointer to the structure holding values that are filled into registers before the interrupt is called. Another parameter was added to select the interrupt number to be called. This way the interface is more universal.

When there is an interrupt function that requires passing or receiving higher amounts of data there was a function implemented that provides the buffer accessible from the real mode. The interrupt function gets the real mode pointer to this buffer in the registers.

Because the real mode can operate under the normal conditions only in the first MB of the memory and the RTEMS image is loaded above this boundary, there are several steps that must be performed before the switch to the real mode occurs:

- Copy values/data that should be passed to the real mode into the first MB of the memory.
- Copy the code that is to be executed in the real mode into the first MB of the memory.
- In the copied code from the previous bullet rewrite the interrupt number with the corresponding parameter passed.

Rewrite of the interrupt number is done in the interrupt instruction to which the interrupt number is bounded. Byte of the interrupt instruction code holding the interrupt number is overwritten by the interrupt number passed to the interface.

In the data copied into the first MB of the memory is included structure holding values of selected registers. The registers are filled with these values prior to the interrupt call.

Register values (return values) after the interrupt call are stored back to the place they were filled from before the interrupt call.

3.3.4 Interface memory layout

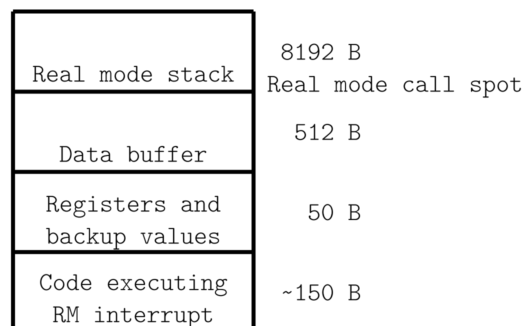


Figure 3.1. Real mode interrupt call memory layout for data and code used.

The interface creates the structure shown on the picture in the first MB of the memory so that these data and the code are accessible while in the real mode. The location of this structure is determined by an address defined by the preprocessor macro. The structure is fully relocatable within the available memory in the first MB of the memory by setting the address of the defined macro to the required value. The top of the stack is located at the address defining the location of the structure. The size of the stack was chosen arbitrarily taking the possible stack usage by graphic BIOSes into consideration. The data buffer size is based on the biggest structure that is passed to/from the VBE. The code part provides the switch to the real mode, the interrupt execution and the switch back to the protected mode.

The first MB is not used by the RTEMS operation system therefore it could be used for the purposes of the interface. The RTEMS is loaded above the first MB of the memory.

3.4 Implemented frame buffer driver

c/src/lib/libbsp/i386/pc386/start/start.S
c/src/lib/libbsp/i386/pc386/include/fb_vesa.h
c/src/lib/libbsp/i386/pc386/console/fb_vesa_rm.c

In this work the real mode interface is used to access VBE functions and it is implemented in the *fb_vesa_rm.c* frame buffer driver. The mode is selected on the boot up. The frame buffer bootup function is called among other start routines in *start.S*. The frame buffer bootup function objective is to set the graphic mode used for the whole executive runtime. Such a bootup function is not a standard way to initialize the frame buffer driver. The reasons for this approach are discussed in the section 2.4.2. The application code later obtains the information about the set mode through the standard way.

3.4.1 VBE interface

Introduction to the VBE showed there are several ways to access its functions. The header file *fb_vesa.h* describing the VBE functions was introduced as an interface to be implemented by the particular VBE frame buffer drivers.

3.4.2 Selecting graphics mode

The graphics mode description obtained using respective VBE function contains besides others these attributes: mode attributes, mode resolution, bits per pixel and physical address for the flat memory frame buffer.

List of graphic mode numbers supported by the graphic card is obtained from the data returned by the adapter info VBE function. A local list of modes with selected fields is created and it is filtered using mode attributes field so that the list contains only modes with these attributes: the mode supported in the hardware, the colored mode (not monochrome), the graphics mode (not text) and the information if the mode has the linear frame buffer available. I sort the list of modes from the highest resolution and the highest bpp to the lowest. The number of pixels in one line has the highest priority followed by the number of lines in one screen image. The bits per pixel field has the lowest priority.

3.4.3 Multiboot options

It is possible to pass so called command line as a multiboot option that can be read by the loaded image and the image can then modify its behavior according to arguments in the command line string. When the command line contains the argument in this format *--video=⟨resX⟩x⟨resY⟩[-bpp]*, the program tries to find the mode from the sorted list that has corresponding parameters. If such a mode is found then it is also set and used.

3.4.4 EDID

Monitors and displays in general contains the EDID structure that defines device capabilities. This structure can be obtained using appropriate sub-function of the Display Data Channel VBE function [10]. If there is no multiboot option *--video=* or no mode with corresponding parameters is found then the EDID structure informations are tested whether at least one mode corresponding to these informations can be found.

The EDID structure may contain four different types of the device timing mode information. Each available timing mode information has defined resolution or resolutions in some way, that is the most relevant information to be used. Types of the timing mode information have the priority order which suggests the order of testing the modes [21].

3.4.5 RTEMS frame buffer

The RTEMS presents the frame buffer to applications as a named device node in the file system. The standard file operations need to be implemented by the higher layer of the VBE support driver. The required functions provided by the frame buffer driver are initialize, open, close, read, write and control. The control function provides the screen informations to the higher layers. Screen informations are kept in two structures. The command parameter of the control function can select the structures to be obtained by the function. One of the two structures contains the screen resolution, bits per pixel and then the information about the division of pixel bits among colors. In the one pixel there are red, green and blue fields. In some cases there is the information about the pixel transparency also specified. Each color has its bit start position in a pixel and the number of bits that a particular color takes in one pixel and the indication whether the most significant bit is on the left or the right side.

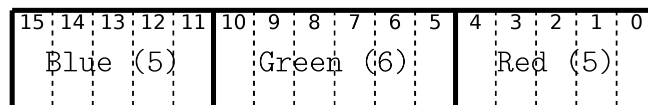


Figure 3.2. Example of packed pixel with 16 bpp. For example offset of green color is the fifth bit and number of its bits (length) is 6. Values for transparent part are in this case equal to zero.

The other structure contains the start address of the frame buffer in the memory, the number of pixel per line, the length of frame buffer memory available, the pixel type (I used packed pixels) and the pixel visual type defining color scheme.

For the implemented driver these structures are set up when the mode is selected which happens in the bootup function. Standard initialize function only registers the frame buffer device while everything else is initialized in the bootup function. The frame buffer control function has also nothing else implemented besides commands for the returning screen info.

The open and the close frame buffer functions uses a mutex to prevent the opening of the frame buffer multiple times.

The last two functions handle the frame buffer content. One function allows to write to the frame buffer and the other to read from it.

Chapter 4
Testing targets and debugging

In order to test the developed software, there had to be a PC machine with the i386 processor available that would run the software. For the early testing the QEMU virtual machine software was used. Later the code was tested on the real hardware. There were used several options to boot the code on real hardware.

When the RTEMS application is compiled there is a bootable image created. This image can be loaded by the target machine through the standard ways that the machine supports.

I tested the new driver code itself with graphical applications, that already run with different frame buffer drivers. The goal was to get a similar graphical output from the application for the old and the new driver.

The testing of the implementation of parts not specific to the driver itself, such as the manipulation with GDT entries, by printing debug messages to the output and by controlling the output for expected messages.

4.1 Virtual hardware

The QEMU utility states on its homepage: "QEMU is a generic and open source machine emulator and virtualizer." [1] In this work the QEMU was used for virtualizing the IBM PC compatible machine. The virtualizer loaded an image containing the new tested RTEMS code.

This work used the QEMU with several arguments:

argument	meaning
-kernel	Specifies multiboot format image to be loaded into the virtualizer.
-vga	Select type of VGA card to emulate.
-s	Starts gdbserver on TCP port 1234.
-S	Do not start CPU at startup.
-append	Parameters to pass as multiboot options.

Table 4.1. Arguments used with QEMU.

4.1.1 GDB

The GNU Debugger tool was used to trace developed program execution and to find spots where the code crashed. This tool contains a rich set of commands. There was also used a graphical front-end for the GDB – the DDD that stands for The Data Display Debugger.

When using one of these utilities it was necessary to connect to the QEMU that runs the debugged code. That is done by the starting of QEMU with the parameter that starts the GDB server on the TCP port.

[1] `http://qemu.org`

In the moment the connection to the GDB server is established, the flow of the debugged program can be controlled by the gdb commands. The list of debugging commands can be obtained by starting the gdb utility and then by running gdb command 'help'.

4.2 Real hardware

There are standard methods of booting up the system on the PC compatible the computers. The priority of booting can be set in computer's 'SETUP'. Even though there are options such as a floppy disk, a CDROM, a hard drive or a flash disk, the development would be significantly slower if these were used for the booting of developed software, because it would require to install a new version of the software on a mentioned medium every time the software changes.

The best option is to use the booting over the network. It is often supported in the BIOS, but many BIOSes I used limit the size of the image that can be loaded. Therefore I used the iPXE[1] bootloader on the target side. I downloaded the image of the iPXE for a flash disk and set up the target PC to boot from that flash disk. Another option would be to load another bootloader which could then load the big sized image. Nevertheless I preferred to use the bootloader on a flash drive for cases where there was no network bootloader on the PC at all. The iPXE can be also loaded from the floppy disk or the CD/DVD if the target PC does not support booting from the flash drive.

I also had to set up a machine where I developed the software that would provide the image that is meant to be run on the target machine. That required to set up the DHCP and the TFTP servers according to the PXE implementation as stated in the guide I found[2]. I used the utility ATFTP as a TFTP server and the ISC DHCP from the Internet Systems Consortium[3].

I added a line to my compile script, that copy the new compiled image to the TFTP default directory and set the DHCP option filename to refer to the image file.

Now every time the new version of tested software were compiled it is sufficient to reboot the target hardware and it loads the image automatically.

Later I also used the tool Novaboot[4], that automates the network booting and makes the booting of remote target with the local OS image simple.

There exists also the option to use GRUB multiboot boot loader to boot the RTEMS executive.

[1] `http://ipxe.org`
[2] `http://www.syslinux.org/wiki/index.php/PXELINUX`
[3] `http://isc.org`
[4] `https://github.com/wentasah/novaboot`

Chapter 5
Integration

5.1 Graphic demos

As stated in the previous chapter the implemented driver was tested with more graphical applications. The RTEMS is used with Microwindows/Nano-X port in more applications and the Nano-X was one of the applications–frameworks used for new driver testing. Demos of the Nano-X were a good way to test the driver by comparing the generated graphical output with the expected one.

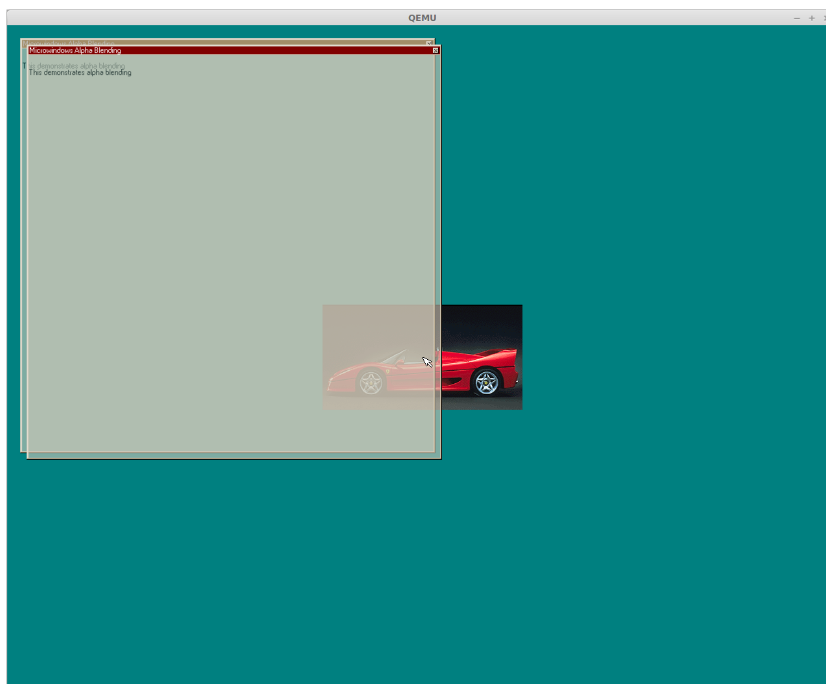


Figure 5.1. Nano-X demo malpha in Qemu with resolution 1280x1024 and 32 bits color depth.

RTEMS project hosts the RTEMS Graphics Toolkit (RGT), which is "collection of free software packages that are suitable for use in building graphical interfaces for RTEMS-based embedded systems"[1]. Microwindows and Nano-X repositories must be cloned into RGT subdirectories. I used versions of nxlib and microwin repositories forked for the RTEMS[2]. The script `do_it` in the RGT root with the parameter `-n` then allows to build Microwindows/Nano-X.

The microwindows repository contains a directory with configurations for systems using it. Config file for the RTEMS `src/Configs/config.rtems` needs to be slightly modified for the driver to work correctly in 32 bit depth modes otherwise there are

[1] `http://git.rtems.org/rtems-graphics-toolkit/tree/README`
[2] `https://github.com/alex-sever-h?tab=repositories`

color channels mixed. Microwindows does not use the information about color channels provided by the RTEMS frame buffer driver, the information must be provided manually in the config file. It only requires to change the pixel type value: `SCREEN_PIXTYPE = MWPF_TRUECOLORABGR`.

The RTEMS config file for Microwindows also needs to be patched to get demos built. There is mistyped check for the existence of the freetype font library. It checks for the tiff library instead. When not patched, the tiff library is found which leads to the freetype compile flag to be set. The Microwindows demos configuration/build does not know how to process such a flag and demos would need to be patched to recognize and process it correctly. Further the config file for Microwindows also set similar flags for the jpeg and the png libraries. The Microwindows/Nano-X demos are not aware of these flags and the compilation fails if the flags are present. The demos do not need these libraries, so to build demos it is necessary to not build or remove the existing jpeg, the freetype and the png libraries. When the libraries are not found by the Mircowindows config file, problematic flags are not set and the compilation succeeds.

5.2 Code

The RTEMS has a tool that checks some of the coding conventions in the rtems-testing repository. The tool should be used for the cleaning of the code before it can be included in the RTEMS mainline. Further the developers which maintain the RTEMS review the code if sent to the RTEMS devel mailing list.

The RTEMS community and maintainers require that the code is accompanied by accurate documentation to be accepted. The RTEMS project uses Doxygen for documenting source code. The Doxygen documentation is written in the source code files as a special comment. Such a documenting comment is bounded to the following lines of the source code.

The slight adjustment of doxygen configuration file was necessary, so that functions and structures using preprocessor macros for compiler attributes (e.g. `typedef struct {...} COMPILER_PACKED my_struct;`) are not shown in the documentation under the macro name (`COMPILER_PACKED`), but under their right names (`my_struct`).

The code was separated into several patches and these were consulted on the RTEMS developer mailing list. After discussion over these patches with the RTEMS community and the adjustment of patches if necessary, RTEMS maintainers pushed the patches to the mainline repository.

Chapter 6
Conclusion

The first approach to the implementation of the VBE driver used the interface of the BIOS of the graphic cards (16-bit protected mode interface) is not well supported or not implemented properly by manufacturers. As my attempt to use this interface failed and as I searched for the reason why, I started to consult other public resources. Apparently, the problem I had with the implementation was not an unusual one since I have found references of the obstacles that other programmers faced as well. It showed me that if the general interface is to be used as defined by the standard, it is necessary to check how manufacturers support that standard in reality. It is also reasonable to check with other sources, if such sources are available, that the features defined by the standard can be actually implemented.

It seemed the protected mode interface is implemented on tested machines, but it turned out that it is not correctly/fully implemented by the manufacturers. Although it might be possible to patch the code of the graphic cards BIOS, in my opinion, the outcome of such an action would be unsure. Instead, I decided to implement or use another interface defined by the VBE – the real mode interface. When this interface has been implemented in the driver it worked on every tested machine. Nevertheless the interface is utilized only on the system startup and the settings of graphic output can not be changed later.

The implemented VBE frame buffer driver for the IBM PC compatible computer now makes it possible to work in the highest resolution mode supported by the graphic card and the monitor. It can be used for the porting of graphic libraries for the RTEMS. This application is well suited for PCs since this type of machine is a widespread one. Moreover, the PC emulators on underlying PC are able to run the native code fast. Applications utilizing real hardware might also appear due to constant lowering power consumption of the i386 chips. These chips may further contain integrated graphic card. This enables us to minimize the size of manufactured devices.

This research enabled me to gain experience with the world wide developers community and the community of open source from the developer point of view. Among other things, I deepened my knowledge of Linux operating system used for the development, the Git version control system, the network booting or the advanced editor Vim.

6.1 Future work

Apparently, there would be a space for the implementation of an another driver which would use a PC code emulator. The driver would then interpret the real mode graphic card BIOS code and that would allow the RTEMS executive to have a full control over its execution. This implies that the calling of the code would be possible anytime.

The new driver should be set as the default one while using the RTEMS on a PC which contains a graphic card. That would require another round of testing and review by the community.

References

[1] VESA. VESA BIOS EXTENSION (VBE): Core functions standard, version: 3.0 [online], 1998.
http://www.cs.utexas.edu/~dahlin/Classes/UGOS/reading/vbe3.pdf.

[2] INTEL. Intel 64 and IA-32 architectures, software developer's manual, combined volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C [online], 2014.
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[3] BRANDEJS Michal. *Mikroprocesory Intel 8086 – 80486 [available online]*. Grada, 1991. ISBN: 80-85424-27-4.

[4] PLUHÁČEK Alois. *8086 - přehled instrukcí*. Česká informatická společnost, 3 edition, 2000. ISBN: 8090085369.

[5] KOZIEROK Charles M. The PC Guide [online], 2001.
http://pcguide.com/ref/ram/umaMap-c.html.

[6] PLUSQUELLIC Jim. Systems design and programming [online].
http://www.ece.unm.edu/~jimp/310/slides/intro_arch.html.

[7] MESSMER Hans-Peter; DEMBOWSKI Klaus. *Hardware: architektura, funkce, programování*. CP Books, a.s., 2005. ISBN: 80-251-0416-8.

[8] SKRAMUSKÝ Robert. *Průvodce nitrem DOSu a BIOSu*. Grada, 1993. ISBN: 80-85623-90-0.

[9] Jhawthorn. OSDev.org: BIOS [online], 2006.
http://wiki.osdev.org/BIOS.

[10] VESA. VESA BIOS Extensions/Display Data Channel Standard, 1999.

[11] VESA. VESA BIOS Extension/Accelerator Functions, 1996.

[12] NOISTERNIG Michael. Operating system programming: VESA VBE 3.0 [online], 2005.
http://notion.muelln-kommune.net/newos_vbe.html.

[13] OAR. Getting started with RTEMS [online], February 2013.
http://docs.rtems.org/doc-current/share/rtems/pdf/started.pdf accessed 2014-12-11.

[14] OAR. Applications c user's guide [online].
http://docs.rtems.org/releases/rtemsdocs-4.7.0/share/rtems/html/c_user/index.html.

[15] OAR. RTEMS development environment guide [online], February 2013.
http://docs.rtems.org/doc-current/share/rtems/pdf/develenv.pdf accessed 2014-12-11.

[16] OAR. BSP and Device Driver Development Guide [online].
http://docs.rtems.org/doc-current/share/rtems/pdf/bsp_howto.pdf.

[17] GNU. GNU binutils [online].
https://www.gnu.org/software/binutils/.

[18] GNU. GCC: Extended asm - assembler instructions with c expression operands [online].
https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[19] GNU. GCC: Constraints for asm operands [online].
https://gcc.gnu.org/onlinedocs/gcc/Constraints.html.

[20] OAR. RTEMS CPU Architecture Supplement [online].
http://docs.rtems.org/doc-current/share/rtems/pdf/cpu_supplement.pdf.

[21] VESA. VESA ENHANCED EXTENDED DISPLAY IDENTIFICATION DATA STANDARD, 2006.
http://ftp.cs.nctu.edu.tw/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_I
EEDIDrAr2.pdf.

[22] GNU. GCC: Specifying attributes of types [online].
https://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html.

[23] GNU. Documentation for binutils 2.24: Using as [online].
https://sourceware.org/binutils/docs-2.24/as/index.html.

[24] SMITH Bob. Transition from protected mode to real mode.
http://www.sudleyplace.com/pmtorm.html.

Appendix A
Specification

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
Katedra měření

Akademický rok **2013-14**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Jan Doležal**
Studijní program: Obor:	**Otevřená informatika** **Počítačové systémy**
Název tématu česky:	**Ovladače a knihovny pro OS RTEMS podporující grafické karty současných PC počítačů**
Název tématu anglicky:	**Drivers and Libraries Enabling RTEMS OS GUI on Current PC Graphics Cards**

Pokyny pro vypracování:

Implementujte a integrujte potřebné komponenty pro podporu grafického výstupu systému RTEMS na současných počítačích třídy PC. Současné grafické karty podporují standard VESA BIOS Extension (VBE), pro který RTEMS driver neexistuje. Výsledkem práce má být funkční ovladač, jeho otestování a integrace s knihovnami pro tvorbu aplikací s grafickým rozhraním pro OS RTEMS.

- Seznamte se s real-time operačním systémem RTEMS a prostředím emulátoru i386/PC (QEMU) pro testování.
- Prostudujte standard VESA BIOS Extension a připravte hlavičkové soubory popisující jeho funkce a struktury (uvažujte licenční podmínky pro integraci projektu RTEMS).
- Implementujte a otestujte ovladač grafických karet s podporou VBE.
- Ozkoušejte s realizovaným ovladačem grafické knihovny Nano-X/Microwindow.
- Zdokumentujte implementovaný software a způsob testování.
- Proveďte potřebné kroky pro začlenění navrženého kódu a úprav do zmíněných projektů a navrhněte úpravy a testování dalších projektů (např. FLTK, SuiTk).

Seznam odborné literatury:

[1] Video Electronics Standards Association: VESA BIOS Extension (VBE), Core Functions Standard 3.0, 1998
[2] RTEMS project documentation and On-Line Library: Applications C User's Guide, POSIX API User's Guide

Vedoucí bakalářské práce:	Ing. Pavel Píša, Ph.D. (K 13135)
Datum zadání bakalářské práce:	2. ledna 2014
Platnost zadání do[1]:	30. ledna 2015

Prof. Ing. Vladimír Haasz, CSc.
vedoucí katedry

Prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 2.1.2014

[1] Platnost zadání je omezena na dobu dvou následujících semestrů.

A.1 Specification in English

Implement and integrate necessary components for the support of the graphical output of the RTEMS operating system on current PC class computers. Present–day graphics cards support the VESA BIOS Extension (VBE) standard but there is no RTEMS driver which implements this standard. The result of this work should be an operational driver, as well as its testing and integration with libraries for the creation of applications with the graphical interface for the RTEMS OS.

- Apprise yourself of the real–time operating system RTEMS and of the environment of the i386/PC emulator (QEMU) for testing.
- Read the VESA BIOS Extension standard and prepare header files describing its functions and structures (consider license requirements for the integration with the RTEMS).
- Implement and test the graphic cards driver supporting the VBE.
- Test graphical libraries Nano-X/Microwindow with the implemented driver.
- Prepare documentation of the implemented software and the form of testing.
- Perform necessary steps for the incorporation of the designed code and adjustments to mentioned projects and suggest adjustments and the testing of the further projects (e.g. FLTK, SuiTk).

Appendix B
Abbreviations

ABI	Application Binary Interface
BIOS	Basic Input/Output System
bpp	Bits Per Pixel
BSP	Board Support Package
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
DDD	The Data Display Debugger
DHCP	Dynamic Host Configuration Protocol
EDID	Extended Display Identification Data
FLTK	Fast, Light Toolkit
GCC	GNU C Compiler
GDB	The GNU Debugger
GNU	GNU's Not Unix!
ISR	Interrupt Service Routine
OAR	On-Line Applications Research Corporation
PC	Personal Computer
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
PXE	Preboot Execution Environment
QEMU	Quick EMUlator
RTEMS	Real Time Executive for Multiprocessing Systems
RTOS	Real Time Operating System
TFTP	Trivaial File Transfer Protocol
VBE	VESA BIOS Extensions
VESA	Video Electronics Standards Association
VGA	Video Graphics Array