Microprocessors Lab
Project

UNIVERSIDADE DE SÃO PAULO • ESCOLA POLITÉCNICA

Prof. Dr. Jorge Kinoshita

Andrei dos Santos - 10333268

Gabriel Kenji Godoy Shimanuki 10336719

Gabriel Pereira de Carvalho - 11257668

February 7, 2024

Contents

1 Abstract

The project proposes an experience involving keyboard data reception. The development environment uses the *toolchain* **gcc-arm-none-eabi** and the system **qemu-system-arm**.

2 Setting up the development environment

First, let's install the tools for our development environment: the *toolchain* **gcc-arm-none-eabi** and the system **qemu-system-arm**.

```
sudo apt-get install gcc-arm-none-eabi
sudo apt-get install qemu-system-arm
```

Note: The activities developed in this manual are executed in a Linux environment and not in the environment used throughout the course (`https://github.com/EpicEric/gcc-arm`).

A *toolchain* is a collection of programming tools that cover everything from program development to generating executable binary files. Typically, a *toolchain* consists of an assembler, a compiler, a linker, a debugger, and, if necessary, auxiliary programs for converting between different types of files.

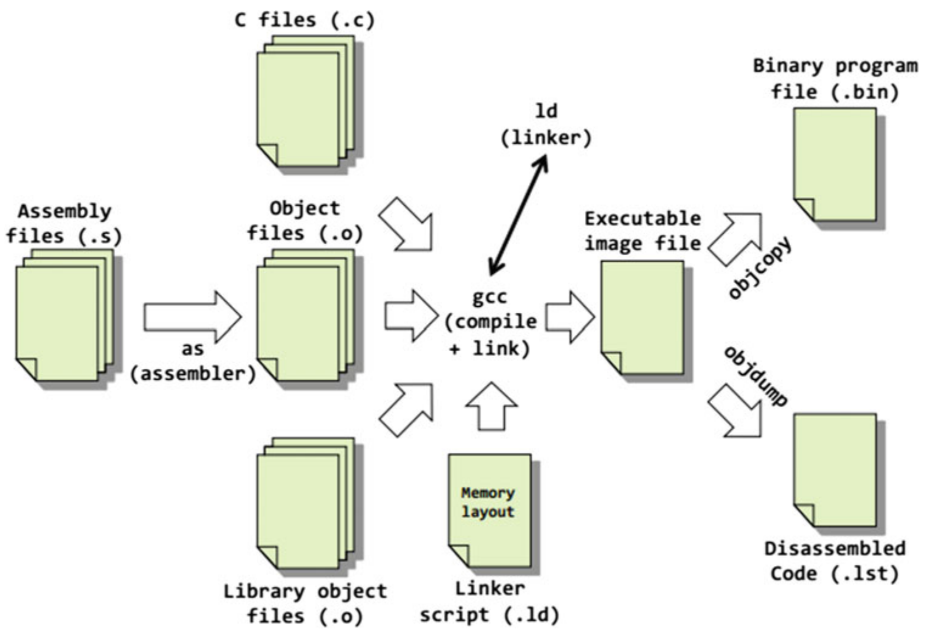


Figure 1: Components of a *toolchain*

The *toolchain* runs on a source system and generates executables for a target system, which in our case is the ARM Versatilepb board (`ARM926EJ-S2016`) that is emulated in the **qemu-system-arm** system.

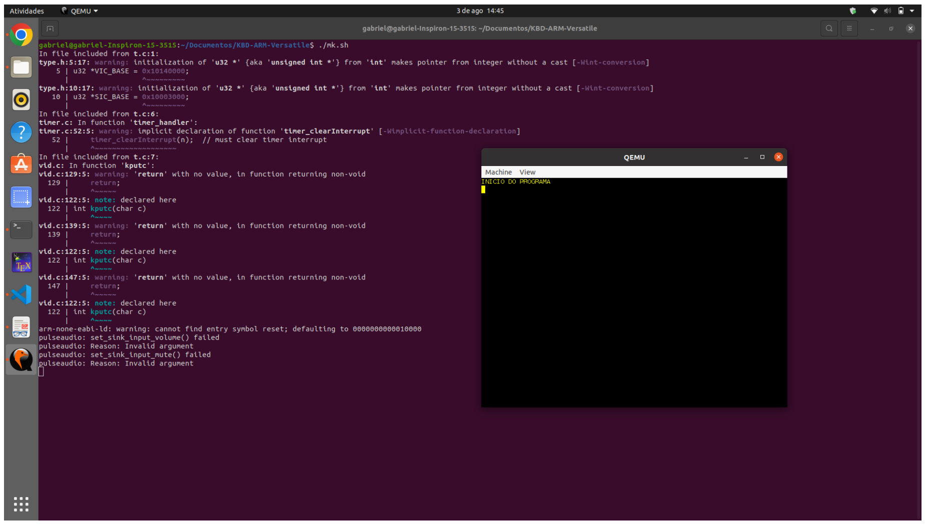
The code to be developed in the practical part of the experiment will be built from a base skeleton that can be found at `https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile`. To download the repository, use the command below.

```
git clone https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile
```

To verify if the repository was downloaded correctly, execute the project by running the *script* `mk.sh`. To do this, enter the directory containing the file.

```
chmod +x mk.sh
./mk.sh
```

If everything is correct, you should see the equivalent of the screen below.


Atividades
QEMU
3 de ago 14:45
gabriel@gabriel-Inspiron-15-3515: ~/Documentos/KBD-ARM-Versatile
QEMU
Machine
View

```
gabriel@gabriel-Inspiron-15-3515:~/Documentos/KBD-ARM-Versatile$ ./mk.sh
In file included from t.c:1:
type.h:5:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
    5 | u32 *VIC_BASE = 0x10140000;
      |                 ^~~~~~~~~~
type.h:10:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
   10 | u32 *SIC_BASE = 0x10003000;
      |                 ^~~~~~~~~~
In file included from t.c:6:
timer.c: In function 'timer_handler':
timer.c:52:5: warning: implicit declaration of function 'timer_clearInterrupt' [-Wimplicit-function-declaration]
   52 |     timer_clearInterrupt(n);  // must clear timer interrupt
      |     ^~~~~~~~~~~~~~~~~~~~
In file included from t.c:7:
vid.c: In function 'kputc':
vid.c:129:5: warning: 'return' with no value, in function returning non-void
  129 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:139:5: warning: 'return' with no value, in function returning non-void
  139 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:147:5: warning: 'return' with no value, in function returning non-void
  147 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
arm-none-eabi-ld: warning: cannot find entry symbol reset; defaulting to 0000000000010000
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
```

```
INICIO DO PROGRAMA
```

Figure 2: Running code from the base repository

3 Introduction to the keyboard driver

The keyboard driver for the ARM Versatile keyboard, or **KBD** (*keyboard driver*), to be developed has three components.

1. an interrupt handler
2. an application program
3. a common data area

When the main program is executed, it is necessary to initialize the keyboard driver control variables in the common data area. When a key is pressed, the hardware generates an interrupt, causing the execution of the interrupt handler.

To handle the interrupt, the interrupt handler must first interpret the typed character. The keyboard has 105 keys, each with its respective scan code. Scan codes up to `0x39` are normal keys, while scan codes above `0x39` are special keys that require different handling.

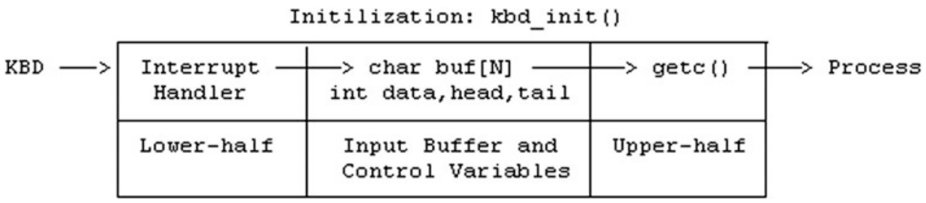


Figure 3: Keyboard driver components

If a normal key has been pressed, the interrupt handler obtains the corresponding `ASCII` value and adds the character to the input buffer of the common data area `buf[N]`. Finally, the interrupt handler notifies the application program that is waiting for keyboard input. The program then retrieves the character value from the buffer.

Note: The driver we are going to develop, based on Wang's book (1), only deals with lowercase characters.

4 Project code

4.1 Project setup

In this section, we will write routines to enable and disable `IRQ` interrupts in `ts.s` and also prepare the conversion tables of `scan codes` to be used by the driver.

In the LAST section, we will develop functions `kgetc()` and `kgets()` of the driver to read variables of types *char* and *string* from the keyboard. During the retrieval of a character from the buffer, it will be necessary to disable `IRQ` interrupts to prevent new modifications to the buffer.

Important: In the terminology learned in **Operating Systems**, the code that modifies the buffer control variables is a critical region. Disabling interrupts prevents race conditions in buffer access.

For this, we will implement `lock` and `unlock` routines in assembly to disable and enable `IRQ` interrupts, respectively. The `lock` function should write `1` to the `I` bit of the `CPSR`, while the `unlock` function should write `0` to the `I` bit of the `CPSR`.

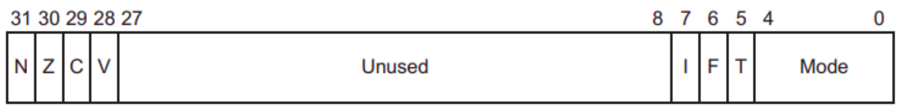


Figure 4: Location of the `I` bit in the `CPSR`

Here is the code to be added to `ts.s`.

```
.global lock, unlock

lock:
  mrs r4, cpsr
  orr r4, r4, #0x80
  msr cpsr, r4
  mov pc, lr

unlock:
  MRS r4, cpsr
  BIC r4, r4, #0x80
  MSR cpsr, r4
  mov pc, lr
```

Before writing the driver, it is necessary to provide a table for conversion between the scan codes provided by the keyboard and the ASCII value of the characters. We will place these tables in the `keymap2` file.

```
//0    1    2    3    4    5    6    7     8    9    A    B    C
     D    E    F
char ltab[] = {
  0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,
     0,   0,   0,
  0,   0,   0,   0,   0,  'q', '1',  0,    0,   0,  'z', 's', 'a',
    'w', '2',  0,
  0,  'c', 'x', 'd', 'e', '4', '3',  0,    0,  ' ', 'v', 'f', 't',
    'r', '5',  0,
  0,  'n', 'b', 'h', 'g', 'y', '6',  0,    0,   0,  'm', 'j', 'u',
    '7', '8',  0,
  0,  ',', 'k', 'i', 'o', '0', '9',  0,    0,  '.', '/', 'l', ';',
    'p', '-',  0,
  0,   0,  '\'',  0,  '[', '=',  0,   0,    0,   0, '\r', ']',  0,
     '\\',  0,   0,
  0,   0,   0,   0,   0,   0,  '\b', 0,    0,   0,   0,   0,   0,
     0,   0,   0
};

char utab[] = {
  0,   0,   0,    0,   0,   0,   0,   0,    0,   0,   0,   0,   0,
      0,   0,   0,
  0,   0,   0,    0,   0,  'Q', '!',  0,    0,   0,  'Z', 'S', 'A', 'W', '@',  0,
  0,  'C', 'X',  'D', 'E', '$', '#',  0,    0,  ' ', 'V', 'F', 'T', 'R', '%',  0,
  0,  'N', 'B',  'H', 'G', 'Y', '^',  0,    0,   0,  'M', 'J', 'U', '&', '*',  0,
  0,  '<', 'K',  'I', 'O', ')', '(',  0,    0,  '>', '?', 'L', ':', 'P', '_',  0,
  0,   0,  '"',   0,  '{', '+',  0,   0,    0,   0,  '\r','}',  0,  '|',  0,   0,
  0,   0,   0,   0,   0,   0,  '\b', 0,    0,   0,   0,   0,   0,   0,   0,   0
};
```

4.2 Keyboard interruption handler

In this section, we will write the two essential routines of the keyboard driver `kbd_init` and `kbd_handler`.
We will develop the `kbd_handler` to print the typed character to the terminal.

Firstly, as we did in the memory-mapped peripherals experiment, we should analyze the keyboard control register.

The keyboard control register is located at address `0x14`. To enable the keyboard, we must set bit 2 of this register, and to configure its use as an Input device, we must set bit 4.

At the beginning of our file, we will import the tables defined in `keymap2` and declare constants with the positions of the relevant bits in the control register.

```
#include "keymap2"

#define KCNTL 0x00
#define KSTAT 0x04
#define KDATA 0x08
#define KCLK  0x0C
```

The basic functionality of the driver involves two functions:

- `kbd_init` responsible for initializing the data structures that we will use and the keyboard control register.
- `kbd_handler` responsible for handling keyboard interrupts.

The essential data structure of the driver is the buffer responsible for storing the typed keys on the keyboard. For buffer management, auxiliary variables are needed.

1. pointer `head` to the start of the buffer.
2. pointer `tail` to the end of the buffer.

3. variable `data` with space already occupied in the buffer.
4. variable `room` with space still free in the buffer.

```c
typedef volatile struct kbd{
  char *base;
  char buf[128];
  int head, tail, data, room;
}KBD;

volatile KBD kbd;

int release;
```

Note: Releasing a key from the keyboard also generates an interrupt with *scode* =`0xF0`, in the interrupt handler we will ignore interrupts with this *scode* value.

Note: Right after releasing the key, an interrupt is generated repeating the character, in the interrupt handler we will ignore interrupts while keeping a global variable `release`.

```c
int kbd_init()
{
  KBD *kp = &kbd;
  kp->base = (char *)0x10006000;
  *(kp->base + KCNTL) = 0x10;    // bit4=activate keyboard bit0=
    activate interrupt
  *(kp->base + KCLK)  = 8;
  kp->head = kp->tail = 0;       // circular buffer char buf[128]
  kp->data = 0; kp->room = 128;
  release = 0;
}

void kbd_handler()
{
  u8 scode, c;
  KBD *kp = &kbd;

  color = YELLOW;

  scode = *(kp->base + KDATA);   // get the character scan code

  if (scode == 0xF0){  // ignore key release
    release = 1;
    return;
  }

  if(release == 1){
    release = 0;
    return;
  }

  // map scan code to ASCII - lowercase
  c = ltab[scode];

  kp->buf[kp->head++] = c;
  kp->head %= 128;
  kp->data++; kp->room--;
}
```

4.3 Basic driver functionality

In this section, we will modify the IRQ interrupt handler and the main program to collect keyboard data.

Now, we need to modify the main program executed from the reset handler, located in the file `t.c`.

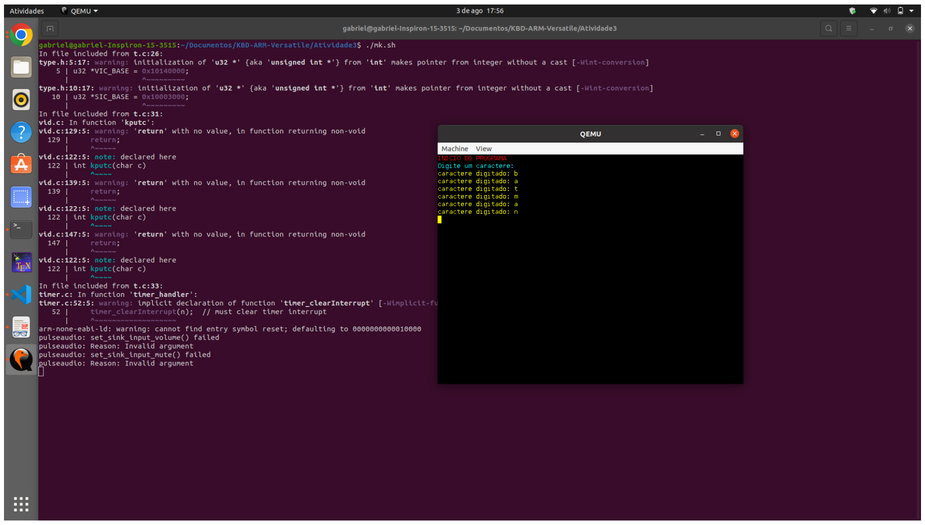
Firstly, we must modify the IRQ interrupt handler to identify keyboard interrupts, differentiating them from timer interrupts. This verification is done in two steps by analyzing the `VIC_STATUS` and `SIC_STATUS` status registers at bits 31 and 3, respectively. If both of these bits are high, it is a keyboard interrupt.

```c
void IRQ_chandler()
{
  int vicstatus = *(VIC_BASE + VIC_STATUS);  // read VIC status
  int sicstatus = *(SIC_BASE + VIC_STATUS);  // read SIC status

  if (vicstatus & (1<<4)){                   // timer0 in bit4 of
    VIC
      timer_handler(0);
  }

  if (vicstatus & (1<<31)){      // VIC bit31 = SIC interrupt
     if (sicstatus & (1<<3)){    // KBD in bit3 of SIC
         kbd_handler();
     }
  }
}
```

In the main routine, we must initialize the keyboard driver by calling the `kbd_init` routine and activate the corresponding keyboard interrupt bits in the `VIC_BASE` and `SIC_BASE` registers.

```c
int main()
{
   color = RED;
   row = col = 0;
   fbuf_init();
   kbd_init();

   *(VIC_BASE + VIC_INTENABLE)  |= (1<<4);          // enable timer
    interrupt
   *(VIC_BASE + VIC_INTENABLE)  |= (1<<31);         // enable
    keyboard interrupt
   *(SIC_BASE + SIC_ENSET)      |= (1<<3);          // enable
    keyboard interrupt

   kputs("PROGRAM START\n");
   timer_init();
   timer_start(0);

   color = CYAN;
   kputs("Enter a character:\n");

   while(1);
}
```

With the modified handler, it will be possible to capture keyboard interrupts during the main program loop. To test the project, simply run the `mk.sh` script.


Atividades
QEMU
3 de ago 17:56
gabriel@gabriel-Inspiron-15-3515: ~/Documentos/KBD-ARM-Versatile/Atividade3
Machine View

```
gabriel@gabriel-Inspiron-15-3515:~/Documentos/KBD-ARM-Versatile/Atividade3$ ./mk.sh
In file included from t.c:26:
type.h:5:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
    5 | u32 *VIC_BASE = 0x10140000;
      |                 ^~~~~~~~~~
type.h:10:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
   10 | u32 *SIC_BASE = 0x10003000;
      |                 ^~~~~~~~~~
In file included from t.c:31:
vid.c: In function 'kputc':
vid.c:129:5: warning: 'return' with no value, in function returning non-void
  129 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:139:5: warning: 'return' with no value, in function returning non-void
  139 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:147:5: warning: 'return' with no value, in function returning non-void
  147 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
In file included from t.c:33:
timer.c: In function 'timer_handler':
timer.c:52:5: warning: implicit declaration of function 'timer_clearInterrupt' [-Wimplicit-fu
   52 |     timer_clearInterrupt(n);  // must clear timer interrupt
      |     ^~~~~~~~~~~~~~~~~~~~
arm-none-eabi-ld: warning: cannot find entry symbol reset; defaulting to 0000000000010000
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
```

```
INICIO DO PROGRAMA
Digite um caractere:
caractere digitado: b
caractere digitado: a
caractere digitado: t
caractere digitado: m
caractere digitado: a
caractere digitado: n
```

Figure 5: Project execution

5 Reading full lines

In this section, we will modify the driver and the main program to capture a line entered instead of just a character.

We will develop two new functions for the driver:

- `kgetc`, which can be called from the main program to capture a `char` input from the keyboard.

- `kgets`, which uses `kgetc` and can be called from the main program to capture a `string` input from the keyboard.

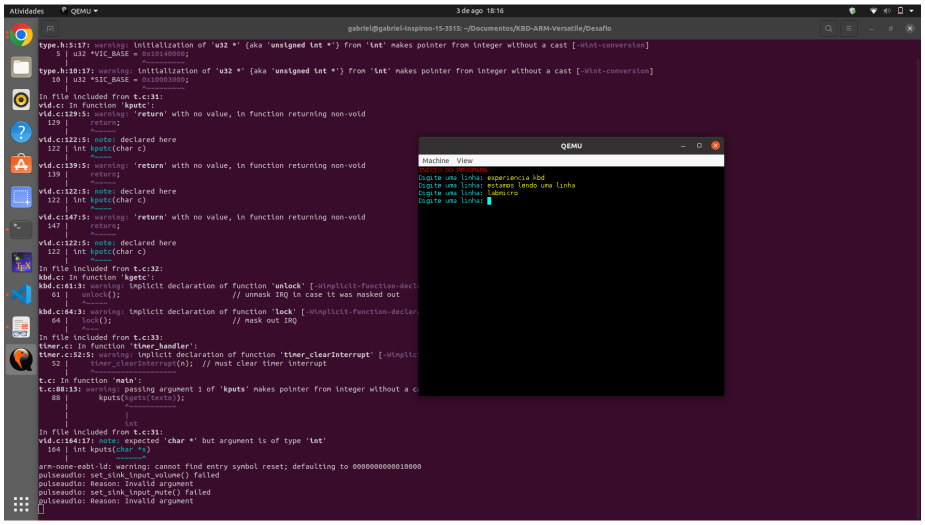
In the `kgetc` function, attention should be paid to the precautions taken to avoid race conditions in buffer access. We have the interruption disablement commented in activity 1 and the use of the **mutex** `kp->data` which keeps the driver in busy wait if necessary.

```c
int kgetc()
{
  char c;
  KBD *kp = &kbd;

  unlock();                        // enable IRQ if disabled
  while(kp->data == 0);            // BUSY wait while kp->data is
     0

  lock();                          // disable IRQ
  c = kp->buf[kp->tail++];
  kp->tail %= 128;                 // Critical Region
  kp->data--; kp->room++;
  unlock();                        // enable IRQ
  return c;
}


int kgets(char s[ ])
{
  char c;
  while( (c = kgetc()) != '\r'){
    kputc(c);
    *s = c;
    s++;
  }
  *s = 0;
}
```

We will also modify `kbd_handler` to stop printing the character immediately after the interruption. We will start printing it in the `kgets` loop as shown in the code above.

For the main program, it is enough to modify the loop where we were waiting for the character input and actively request a new line at each iteration with the `kgets` function.

```c
   while(1){
      color = CYAN;
      kputs("Enter a line: ");
      kputs(kgets(text));
      kputs("\n");
   }
```


Atividades
QEMU
3 de ago 18:16
gabriel@gabriel-Inspiron-15-3515: ~/Documentos/KBD-ARM-Versatile/Desafio
QEMU
Machine
View

```
type.h:5:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
    5 | u32 *VIC_BASE = 0x10140000;
      |                 ^~~~~~~~~~
type.h:10:17: warning: initialization of 'u32 *' {aka 'unsigned int *'} from 'int' makes pointer from integer without a cast [-Wint-conversion]
   10 | u32 *SIC_BASE = 0x10003000;
      |                 ^~~~~~~~~~
In file included from t.c:31:
vid.c: In function 'kputc':
vid.c:129:5: warning: 'return' with no value, in function returning non-void
  129 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:139:5: warning: 'return' with no value, in function returning non-void
  139 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
vid.c:147:5: warning: 'return' with no value, in function returning non-void
  147 |     return;
      |     ^~~~~~
vid.c:122:5: note: declared here
  122 | int kputc(char c)
      |     ^~~~~
In file included from t.c:32:
kbd.c: In function 'kgetc':
kbd.c:61:3: warning: implicit declaration of function 'unlock' [-Wimplicit-function-decl
   61 |   unlock();                           // unmask IRQ in case it was masked out
      |   ^~~~~~
kbd.c:64:3: warning: implicit declaration of function 'lock' [-Wimplicit-function-declar
   64 |   lock();                             // mask out IRQ
      |   ^~~~
In file included from t.c:33:
timer.c: In function 'timer_handler':
timer.c:52:5: warning: implicit declaration of function 'timer_clearInterrupt' [-Wimplic
   52 |     timer_clearInterrupt(n);  // must clear timer interrupt
      |     ^~~~~~~~~~~~~~~~~~~~
t.c: In function 'main':
t.c:88:13: warning: passing argument 1 of 'kputs' makes pointer from integer without a c
   88 |       kputs(kgets(texto));
      |             ^~~~~~~~~~~~
      |             |
      |             int
In file included from t.c:31:
vid.c:164:17: note: expected 'char *' but argument is of type 'int'
  164 | int kputs(char *s)
      |           ~~~~~~^
arm-none-eabi-ld: warning: cannot find entry symbol reset; defaulting to 0000000000010000
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
```

```
INICIO DO PROGRAMA
Digite uma linha: experiencia kbd
Digite uma linha: estamos lendo uma linha
Digite uma linha: labmicro
Digite uma linha:
```

Figure 6: Execution of the challenge

References

[1] K.C. Wang. **Embedded and Real-Time Operating Systems**. Editora Springer (2017).