
Laboratório de Microprocessadores
Projeto



Prof. Dr. Jorge Kinoshita

Bancada B6

Andrei dos Santos - 10333268

Gabriel Kenji Godoy Shimanuki 10336719

Gabriel Pereira de Carvalho - 11257668

4 de agosto de 2022

Sumário

1	Resumo	3
2	Objetivos	3
3	Preparação ambiente de desenvolvimento	3
4	Introdução ao <i>driver</i> para o teclado	5
5	PARTE EXPERIMENTAL	6
5.1	Atividade 1: Modificações em <code>ts.s</code>	6
5.2	Atividade 2: Driver <code>kbd.c</code>	8
5.3	Atividade 3: Programa principal	10
6	Desafio	11

1 Resumo

O projeto propõe uma experiência que envolve a recepção de dados do teclado. Para isso serão utilizadas a *toolchain* **gcc-arm-none-eabi** e o sistema **qemu-system-arm** já abordados ao longo da disciplina.

2 Objetivos

Após a conclusão dessa experiência, os seguintes tópicos devem ser conhecidos pelos alunos:

- Montar arquivos executáveis com a *toolchain* **gcc-arm-none-eabi**.
- Executar códigos na placa *Versatile* emulada em **qemu-system-arm**.
- Construir *driver* para o teclado ARM Versatile.

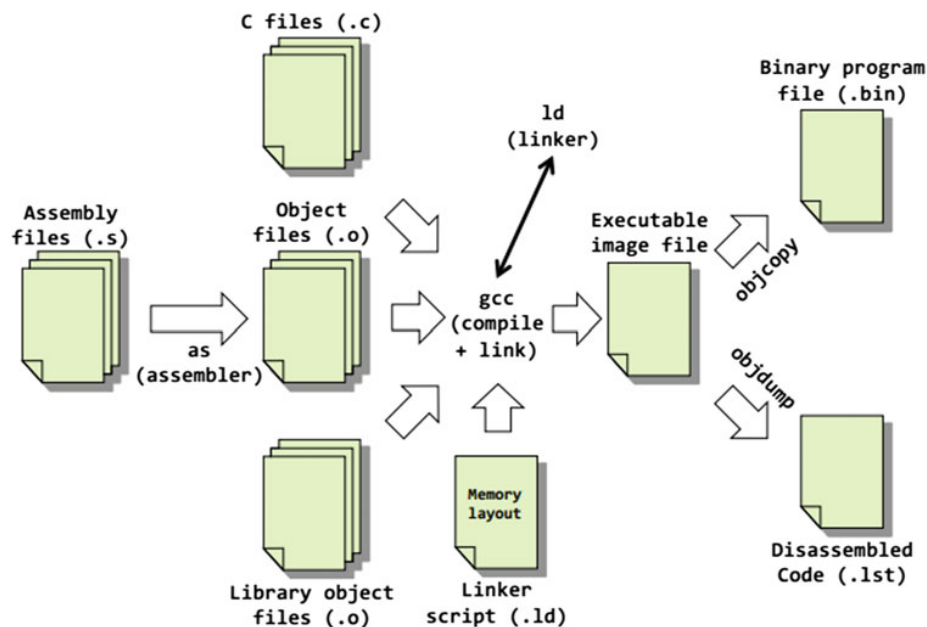
3 Preparação ambiente de desenvolvimento

Primeiramente, vamos instalar as ferramentas de nosso ambiente de desenvolvimento: a *toolchain* **gcc-arm-none-eabi** e o sistema **qemu-system-arm**.

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

Observação: As atividades desenvolvidas nessa apostila são executadas num ambiente linux e não no ambiente utilizado ao longo da disciplina (<https://github.com/EpicEric/gcc-arm>).

Uma *toolchain* é uma coleção de ferramentas de programação que abrangem desde o desenvolvimento de programas até a geração de arquivos binários executáveis. Geralmente, uma *toolchain* é composta por um *assembler*, um *compilador*, um *linker*, um *debugger* e, caso necessário, programas auxiliares para conversão entre tipos diferentes de arquivos.

Figura 1: Componentes de uma *toolchain*

A *toolchain* é executado num sistema de origem e gera executáveis para um sistema destino, que nosso caso é a placa ARM Versatilepb (ARM926EJ-S2016) que é emulada no sistema **qemu-system-arm**.

O código a ser desenvolvido na parte prática da experiência será construído a partir de um esqueleto base que pode ser encontrado em <https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile>. Para baixar o repositório, utilize o comando abaixo.

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

Para verificar se o repositório foi baixado corretamente, execute o projeto executando o *script* **mk.sh**. Para isso entre no diretório que contém o arquivo.

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

Se tudo estiver correto, você deve visualizar o equivalente à tela abaixo.

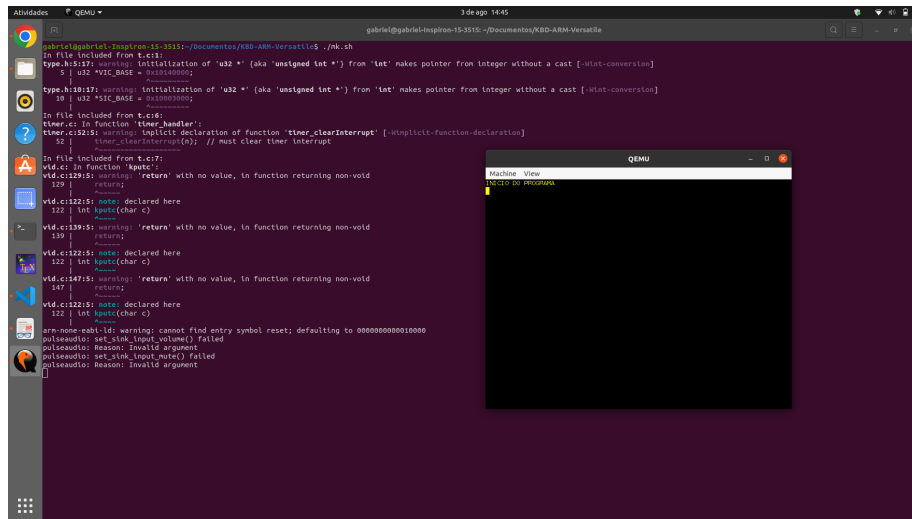


Figura 2: Executando código do repositório base

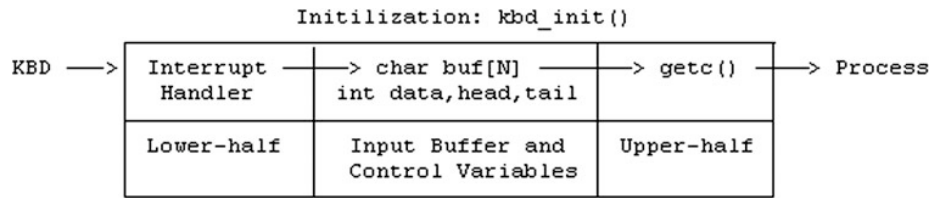
4 Introdução ao *driver* para o teclado

O *driver* para o teclado ARM Versatile, ou **KBD** (*keyboard driver*), a ser desenvolvido possui três componentes.

1. um *interrupt handler*
2. um *application program*
3. uma área comum de dados

Quando o programa principal é executado, é necessário inicializar as variáveis de controle do *driver* do teclado na área comum de dados. Quando uma tecla é pressionada, o *hardware* gera uma interrupção, causando a execução do *interrupt handler*.

Para tratar a interrupção, o *interrupt handler* deve primeiramente interpretar o caractere digitado. O teclado possui 105 teclas, cada uma com seu respectivo *scan code*. Os *scan codes* até 0x39 são teclas normais, enquanto os *scan codes* acima de 0x39 são teclas especiais que requerem um tratamento diferenciado.

Figura 3: Componentes do *driver* do teclado

Caso uma tecla normal tenha sido pressionada, o *interrupt handler* obtém o valor ASCII correspondente e acrescenta o caractere ao *input buffer* da área comum de dados `buf[N]`. Finalmente, o *interrupt handler* notifica o *application program* que está aguardando por entrada do teclado. O programa então irá recuperar o valor do caractere a partir do *buffer*.

Observação: o *driver* que vamos desenvolver, baseado no livro do Wang(1), lida apenas com caracteres minúsculos.

5 PARTE EXPERIMENTAL

5.1 Atividade 1: Modificações em `ts.s`

Nessa atividade, vamos escrever rotinas para habilitar e desabilitar interrupções IRQ em `ts.s` e também preparar as tabelas de conversão de `scan codes` a serem usadas pelo *driver*.

Na seção desafio, vamos desenvolver funções `kgetc()` e `kgets()` do *driver* para ler variáveis dos tipos *char* e *string* a partir do teclado. Durante a recuperação de um caractere do *buffer*, será necessário desabilitar interrupções IRQ para impedir novas modificações no *buffer*.

Importante: na terminologia aprendida em **Sistemas Operacionais**, o código que modifica as variáveis de controle do *buffer* é uma região crítica. Desabilitar interrupções impede condições de corrida no acesso ao *buffer*.

Para isso, vamos implementar rotinas `lock` e `unlock` em *assembly* para desabilitar e habilitar interrupções IRQ, respectivamente. A função `lock` deve escrever 1 no *bit I* do CPSR, enquanto a função `unlock` deve escrever 0 no *bit I* do CPSR.

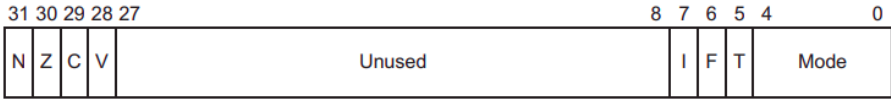


Figura 4: Localização do bit I no CPSR

Segue o código a ser adicionado em `ts.s`.

```

1  .global lock, unlock
2
3  lock:
4      mrs r4, cpsr
5      orr r4, r4, #0x80
6      msr cpsr, r4
7      mov pc, lr
8
9  unlock:
10     MRS r4, cpsr
11     BIC r4, r4, #0x80
12     MSR cpsr, r4
13     mov pc, lr

```

Antes de escrever o *driver*, é necessário fornecer uma tabela para conversão entre os *scan codes* fornecidos pelo teclado e o valor ASCII dos caracteres. Vamos colocar essas tabelas no arquivo `keymap2`.

```

1 //0      1      2      3      4      5      6      7      8      9      A      B      C
2 char ltab[] = {
3     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
4     0, 0, 0, 0, 0, 'q', '1', 0, 0, 0, 'z', 's', 'a',
5     'w', '2', 0, 0, 'c', 'x', 'd', 'e', '4', '3', 0, 0, ' ', 'v', 'f', 't',
6     'r', '5', 0, 0, 'n', 'b', 'h', 'g', 'y', '6', 0, 0, 0, 'm', 'j', 'u',
7     '7', '8', 0, 0, ' ', ' ', 'k', 'i', 'o', '0', '9', 0, 0, '.', '/', 'l', ';',
8     'p', '-', 0, 0, 0, ' ', '=', 0, 0, 0, 0, '\r', ']', 0,
9     '\\\', 0, 0, 0, 0, 0, 0, '\b', 0, 0, 0, 0, 0,
10    0, 0, 0
11 };
12 char utab[] = {
13     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
14     0, 0, 0, 0, 0, 'Q', '!', 0, 0, 0, 'Z', 'S', 'A',
15     'W', '@', 0, 0, 'C', 'X', 'D', 'E', '$', '#', 0, 0, ' ', 'V', 'F', 'T',
16     'R', '%', 0, 0, 'N', 'B', 'H', 'G', 'Y', '^', 0, 0, 0, 'M', 'J', 'U',
17     '&', '*', 0, 0, '<', 'K', 'I', 'O', ')', '(', 0, 0, '>', '?', 'L', ':',
18     'P', '-', 0,

```

```

18 0, 0, '"', 0, '{', '+', 0, 0, 0, 0, '\r', '}', 0,
19 '|', 0, 0, 0, 0, 0, 0, '\b', 0, 0, 0, 0, 0,
20 0, 0, 0
};

```

Para conferir a organização dos arquivos do repositório após essa etapa, consultar <https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile/Atividade1>.

5.2 Atividade 2: Driver kbd.c

Nessa atividade, vamos escrever as duas rotinas essenciais do *driver* do teclado `kbd_init` e `kbd_handler`. Vamos desenvolver o `kbd_handler` para imprimir no terminal o caractere digitado.

Primeiramente, assim como fizemos na experiência de periféricos mapeados na memória, devemos analisar o registrador de controle do teclado.

O registrador de controle do teclado está localizado no endereço `0x14`. Para habilitar o teclado devemos ativar o bit 2 desse registrador e para configurar seu uso como dispositivo de *Input* devemos ativar o bit 4.

No início de nosso arquivo, iremos importar as tabelas definidas em `keymap2` e declarar constantes com as posições dos bits relevantes no registrador de controle.

```

1 #include "keymap2"
2
3 #define KCNTL 0x00
4 #define KSTAT 0x04
5 #define KDATA 0x08
6 #define KCLK 0x0C

```

A funcionalidade básica do driver envolve duas funções:

- `kbd_init` responsável por inicializar as estruturas de dados que iremos utilizar e o registrador de controle do teclado.
- `kbd_handler` responsável por tratar interrupções de teclado.

A estrutura de dados essencial do *driver* é o *buffer* responsável por armazenar as teclas digitadas no teclado. Para gerenciamento do *buffer* são necessárias variáveis auxiliares.

1. ponteiro `head` para início do *buffer*.
2. ponteiro `tail` para final do *buffer*.
3. variável `data` com espaço já ocupado no *buffer*.
4. variável `room` com espaço ainda livre no *buffer*.


```

1 typedef volatile struct kbd{
2     char *base;
3     char buf[128];
4     int head, tail, data, room;
5 }KBD;
6
7 volatile KBD kbd;
8
9 int release;

```

Observação: a liberação de uma tecla do teclado também gera uma interrupção com *scode* = 0xF0, no *interrupt handler* vamos ignorar interrupções com esse valor de *scode*.

Observação: logo após a liberação da tecla é gerada uma interrupção repetindo o caractere, no *interrupt handler* vamos ignorar interrupções mantendo uma variável global *release*.

```

1 int kbd_init()
2 {
3     KBD *kp = &kbd;
4     kp->base = (char *)0x10006000;
5     *(kp->base + KCNTL) = 0x10; // bit4=Enable bit0=INT on
6     *(kp->base + KCLK) = 8; // ARM manual says clock=8
7     kp->head = kp->tail = 0; // circular buffer char buf[128]
8     kp->data = 0; kp->room = 128;
9     release = 0;
10 }
11
12 void kbd_handler()
13 {
14     u8 scode, c;
15     KBD *kp = &kbd;
16
17     color = YELLOW;
18
19     scode = *(kp->base + KDATA); // get scan code of this interrpt
20
21     if (scode == 0xF0){ // ignora liberacao de tecla
22         release = 1;
23         return;
24     }
25
26     if(release == 1){
27         release = 0;
28         return;
29     }
30
31     // map scode to ASCII in lowercase
32     c = ltab[scode];
33
34     kp->buf[kp->head++] = c;

```

```

35 kp->head %= 128;
36 kp->data++; kp->room--;
37 }

```

Para conferir a organização dos arquivos do repositório após essa etapa, consultar <https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile/Atividade2>.

5.3 Atividade 3: Programa principal

Nessa atividade vamos modificar o *handler* de interrupções IRQ e o programa principal para coletar dados do teclado.

Agora, precisamos modificar o programa principal executado a partir do *reset handler*, localizado no arquivo `t.c`.

Primeiramente, devemos modificar o *handler* de interrupções IRQ para identificar interrupções de teclado, diferenciando-as das interrupções de *timer*. Essa verificação é feita em duas etapas analisando os registradores de *status* VIC_STATUS e SIC_STATUS nos bits 31 e 3, respectivamente. Caso ambos estes bits estejam em alto, se trata de uma interrupção de teclado.

```

1 void IRQ_chandler()
2 {
3     int vicstatus = *(VIC_BASE + VIC_STATUS); // read VIC status
4     int sicstatus = *(SIC_BASE + VIC_STATUS); // read SIC status
5
6     if (vicstatus & (1<<4)){ // timer0 at bit4 of
7         VIC
8         timer_handler(0);
9     }
10
11     if (vicstatus & (1<<31)){ // VIC bit31 = SIC interrupts
12         if (sicstatus & (1<<3)){ // KBD at bit3 on SIC
13             kbd_handler();
14         }
15 }

```

Na rotina principal, devemos inicializar o *driver* do teclado chamando a rotina `kbd_init` e ativar os bits correspondentes às interrupções de teclado nos registradores VIC_BASE e SIC_BASE.

```

1 int main()
2 {
3     color = RED;
4     row = col = 0;
5     fbuf_init();
6     kbd_init();
7
8     *(VIC_BASE + VIC_INTENABLE) |= (1<<4); // VIC route
9     timer0 at bit4
10    *(VIC_BASE + VIC_INTENABLE) |= (1<<31); // VIC rount
11    bit31
12    *(SIC_BASE + SIC_ENSET) |= (1<<3); // SIC bit3 = KBD
13    interrupts

```

```

11
12     kputs("INICIO DO PROGRAMA\n");
13     timer_init();
14     timer_start(0);
15
16     color = CYAN;
17     kputs("Digite um caractere:\n");
18
19     while(1);
20 }

```

Com o *handler* modificado, será possível capturar interrupções de teclado durante o *loop* do programa principal. Para testar o projeto, basta rodar o *script* `mk.sh`.

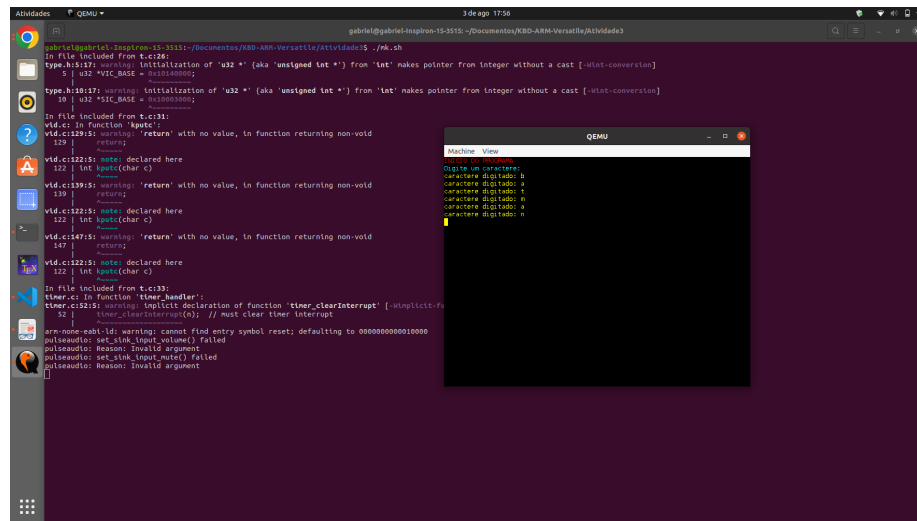


Figura 5: Execução do projeto

Para conferir a organização dos arquivos do repositório após essa etapa, consultar <https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile/Atividade3>.

6 Desafio

No desafio, vamos modificar o *driver* e o programa principal para capturar uma linha digitada e não apenas um caractere.

Vamos desenvolver duas novas funções para o *driver*:

- `kgetc` que pode ser chamada do programa principal para capturar uma entrada tipo `char` do teclado.

- `kgets` que usa `kgetc` e pode ser chamada do programa principal para capturar uma entrada tipo `string` do teclado.

Na função `kgetc` vale chamar atenção aos cuidados tomados para evitar condições de corrida no acesso ao *buffer*. Temos a desabilitação de interrupções comentada na atividade 1 e o uso do **mutex** `kp->data` que mantém o *driver* em *busy wait* caso necessário.

```

1 int kgetc()
2 {
3     char c;
4     KBD *kp = &kbd;
5
6     unlock();                                // unmask IRQ in case it was
7     masked out                               // BUSY wait while kp->data is
8     0
9
10    lock();                                // mask out IRQ
11    c = kp->buf[kp->tail++];
12    kp->tail %= 128;                         /** Critical Region */
13    kp->data--; kp->room++;
14    unlock();                               // unmask IRQ
15    return c;
16 }
17
18 int kgets(char s[ ])
19 {
20     char c;
21     while( (c = kgetc()) != '\r'){
22         kputc(c);
23         *s = c;
24         s++;
25     }
26     *s = 0;
27 }

```

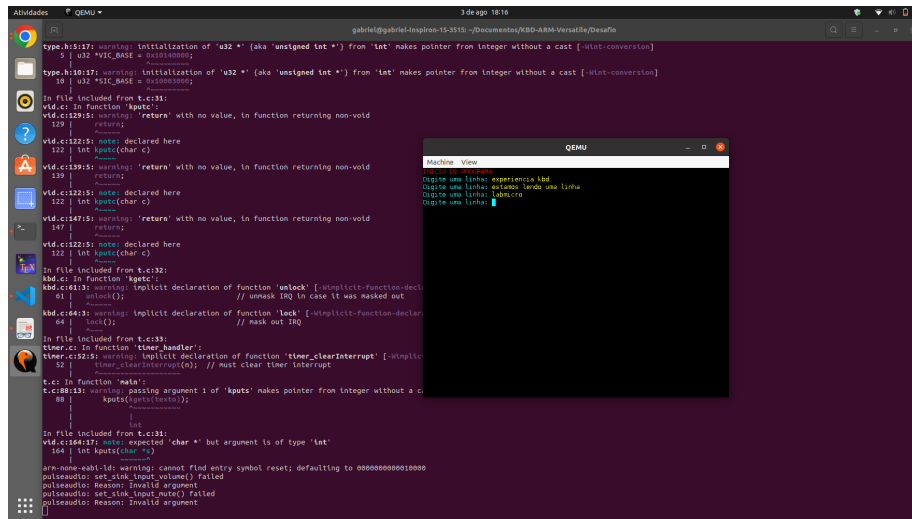
Vamos modificar também `kbd_handler` para deixar de imprimir o caractere logo após a interrupção. Passaremos a realizar essa impressão no *loop* de `kgets` como pode ser visto no código acima.

Para o programa principal basta modificar o *loop* onde aguardávamos a entrada de caracter e ativamente pedir por uma nova linha a cada iteração com a função `kgets`.

```

1 while(1){
2     color = CYAN;
3     kputs("Digite uma linha: ");
4     kputs(kgets(texto));
5     kputs("\n");
6 }

```



```
Abilidades QEMU 3 de ago 18:16
gabriel@gabriel-inspiron-15-3515: ~/Documents/KBD-ARM-Versatile/Desafio
type.h:517: warning: initialization of 'u32 *' (aka 'unsigned int *') from 'int' makes pointer from integer without a cast [-Wint-conversion]
5 | u32 *VIC_BASE = 0x3f100000;
  | ~~~~~^~~~~~
type.h:517: warning: initialization of 'u32 *' (aka 'unsigned int *') from 'int' makes pointer from integer without a cast [-Wint-conversion]
15 | u32 *VIC_BASE = 0x3f100000;
   | ~~~~~^~~~~~
In file included from t.c:131:
vld.c: In function 'kputs':
vld.c:129:9: warning: 'return' with no value, in function returning non-void
129 |     return;
      |     ^~~~~~
vld.c:132:15: note: declared here
132 | int kputs(char c)
      |       ^
vld.c:139:9: warning: 'return' with no value, in function returning non-void
139 |     return;
      |     ^~~~~~
vld.c:132:15: note: declared here
132 | int kputs(char c)
      |       ^
vld.c:147:9: warning: 'return' with no value, in function returning non-void
147 |     return;
      |     ^~~~~~
vld.c:132:15: note: declared here
132 | int kputs(char c)
      |       ^
In file included from t.c:132:
khd.c: In function 'kputc':
khd.c:61:13: warning: implicit declaration of function 'unlock' [-Wimplicit-function-decl]
61 |     unlock();
      |     ^~~~~~
khd.c:64:13: warning: implicit declaration of function 'lock' [-Wimplicit-function-decl]
64 |     lock();
      |     ^~~~
In file included from t.c:133:
timer.c: In function 'timer_handler':
timer.c:52:13: warning: implicit declaration of function 'timer_clearInterrupt' [-Wimplicit-declaration]
52 |     timer_clearInterrupt(0); // Must clear timer interrupt
      |     ^~~~~~~~~~~~~~~~~~~~~
t.c: In function 'main':
t.c:138:13: warning: passing argument 1 of 'kputs' makes pointer from integer without a cast [-Wint-conversion]
138 |     kputs(VIC_BASE);
      |     ~~~~~^~~~~~
In file included from t.c:131:
vld.c:132:15: note: expected 'char *' but argument is of type 'int'
132 | int kputs(char c)
      |       ^
arm-none-eabi-ld: warning: cannot find entry symbol reset; defaulting to 000000000010000
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
```

Figura 6: Execução do desafio

Para conferir a organização dos arquivos do repositório após essa etapa, consultar <https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile/Desafio>.

Referências

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