

PCS3732 - LABORATÓRIO DE PROCESSADORES

Interrupção de Teclado

Andrei dos Santos
Gabriel Kenji Godoy Shimanuki
Gabriel Pereira de Carvalho

Agenda { }

Setup

Parte 1

Preparação do Ambiente de Desenvolvimento

Introdução

Parte 2

Introdução ao driver do teclado

Atividades

Parte 4

Driver do teclado

Parte 3

Rotinas lock e unlock

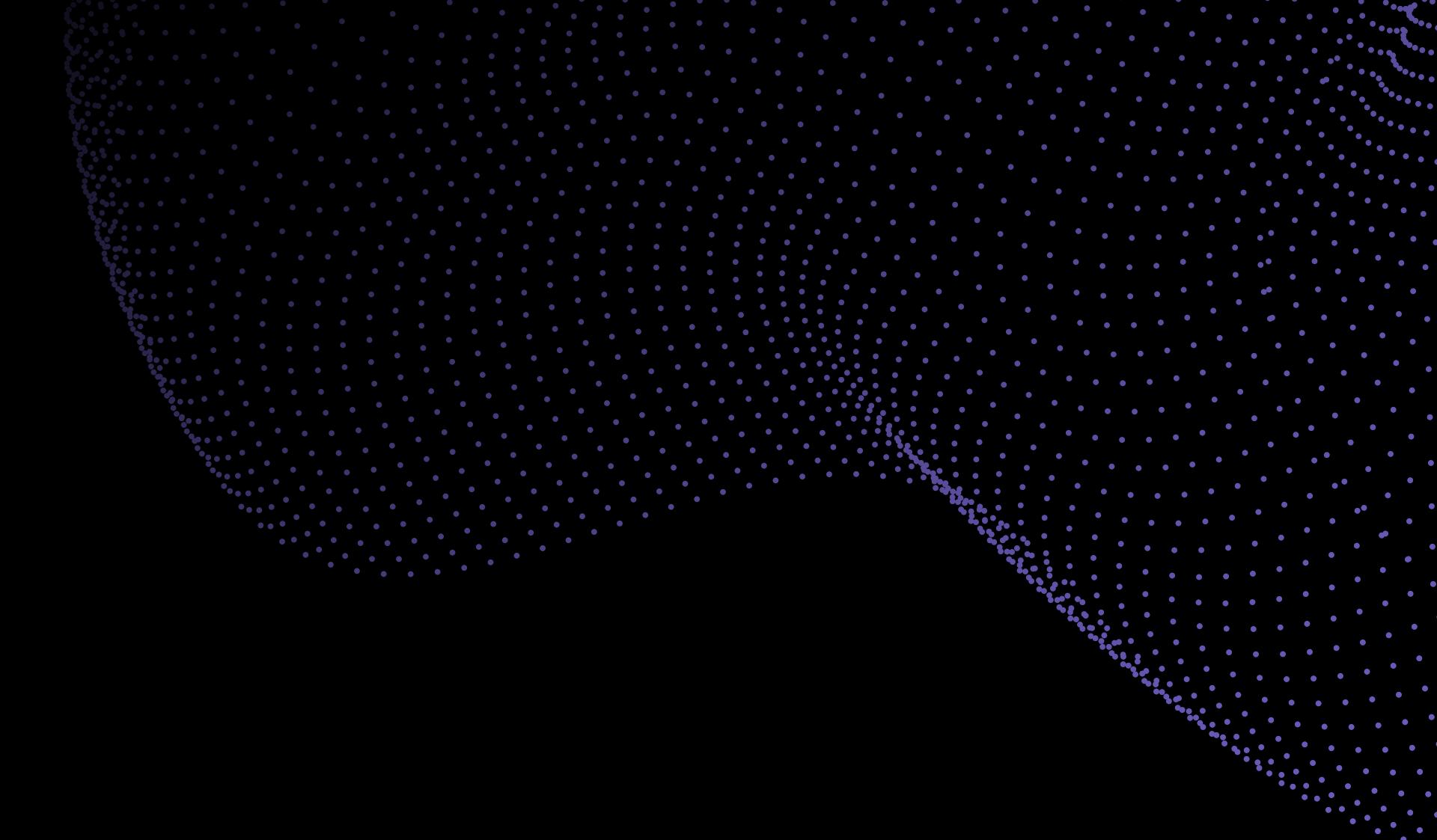
Parte 5

Modificação do programa principal

Desafio

Parte 6

Capturar linha



Parte 1 { }

Setup

Parte 1

Preparação do
Ambiente de
Desenvolvimento

Introdução

Parte 2

Introdução ao
driver do teclado

Atividades

Parte 4

Driver do teclado

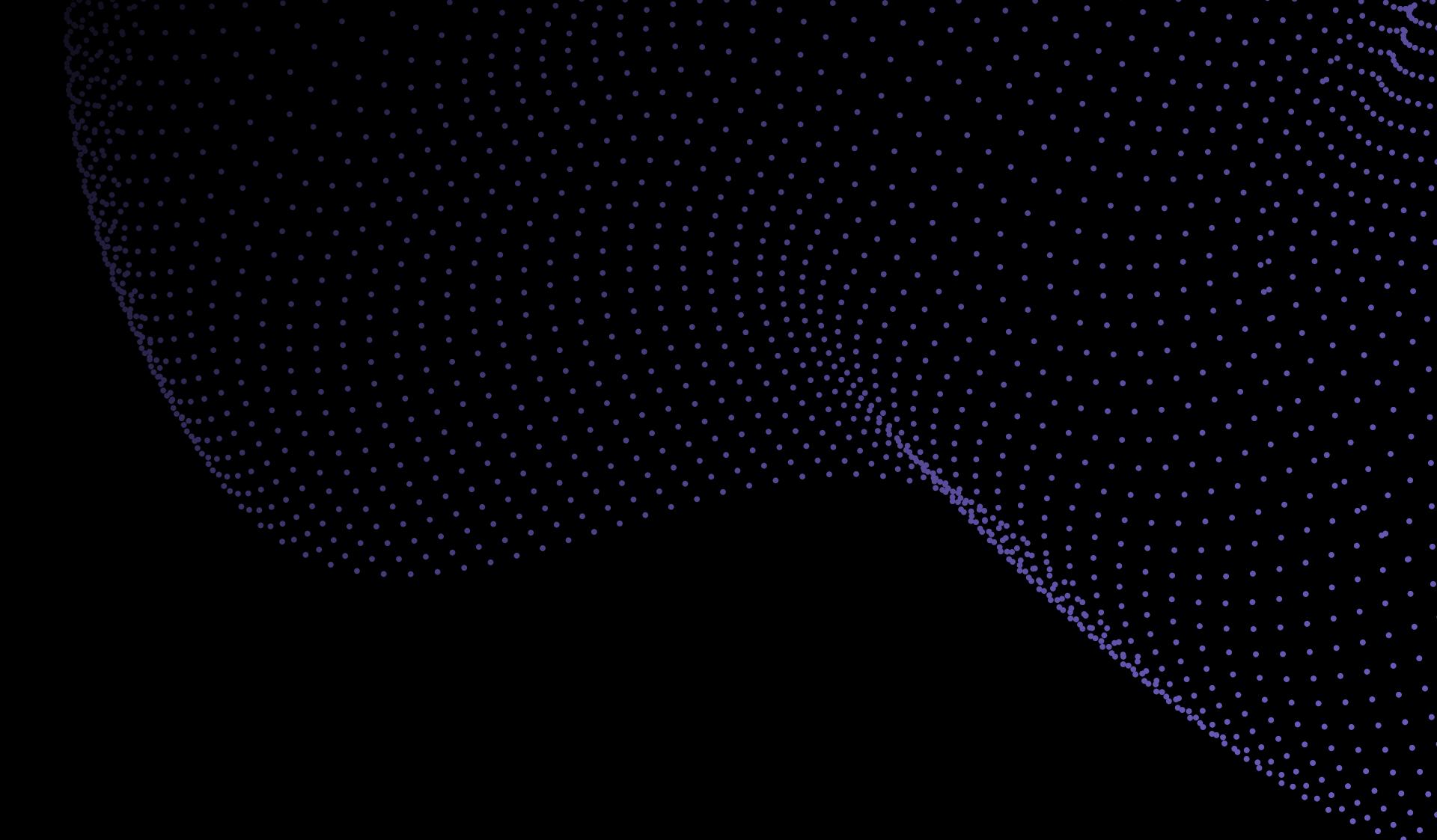
Parte 5

Modificação do
programa
principal

Desafio

Parte 6

Capturar linha

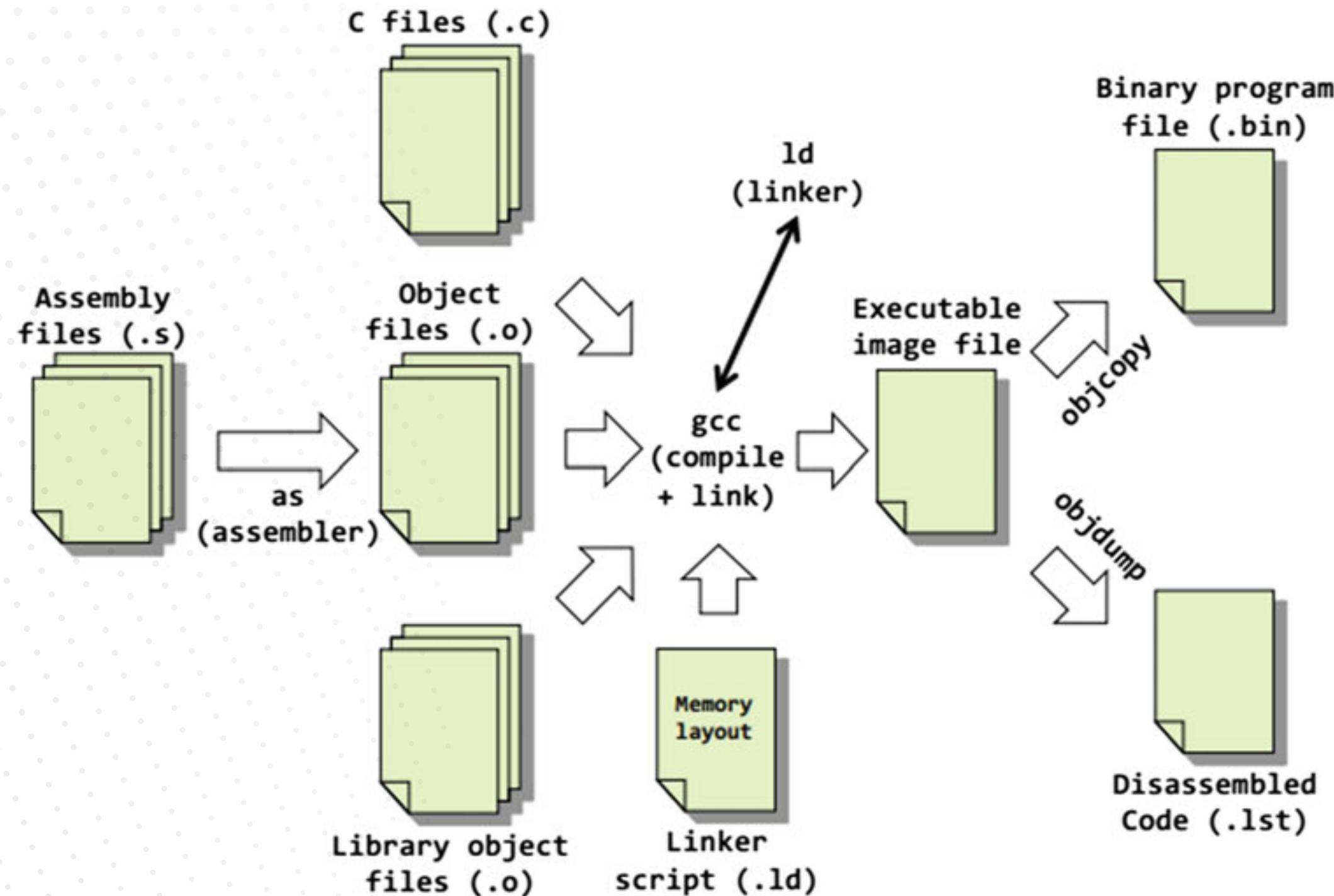


Instalação de Ferramentas

Instalação das ferramentas do ambiente de desenvolvimento:

- **toolchain: gcc-arm-none-eabi**
 - sudo apt-get install gcc-arm-none-eabi
- **sistema: qemu-system-arm**
 - sudo apt-get install qemu-system-arm

Toolchain



Repositório do Projeto

O código desenvolvido é construído a partir do esqueleto base disponível em:

<https://github.com/ArkhamKnightGPC/KBD-ARM-Versatile>

Para baixar o repositório:

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

Para verificar se o repositório foi baixado corretamente:

```
chmod +x mk.sh (altera permissão)  
./mk.sh (executa)
```

Parte 2 { }

Setup

Introdução

Atividades

Desafio

Parte 1

Preparação do
Ambiente de
Desenvolvimento

Parte 2

Introdução ao
driver do teclado

Parte 3

Rotinas lock e
unlock

Parte 4

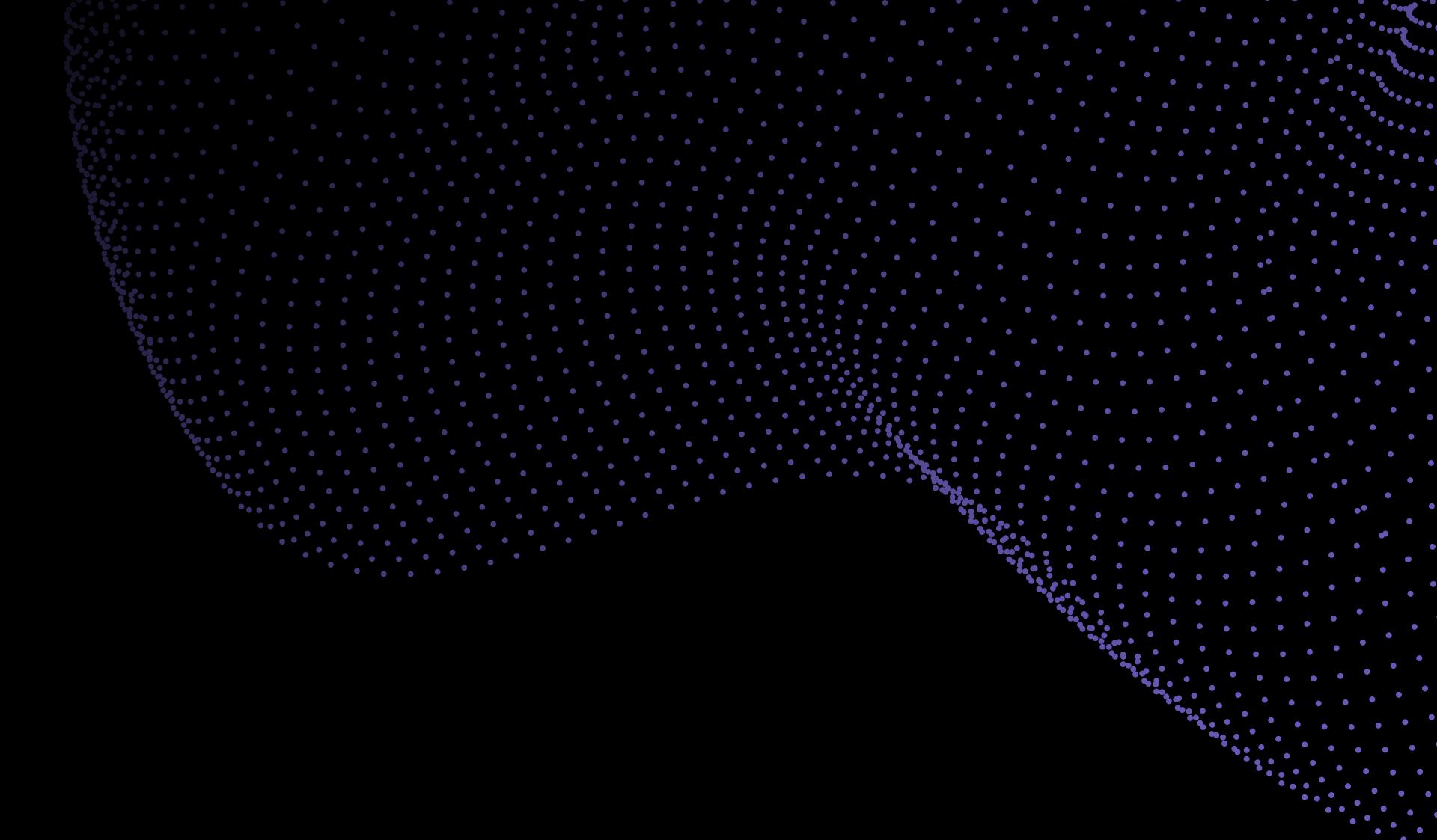
Driver do teclado

Parte 5

Modificação do
programa
principal

Parte 6

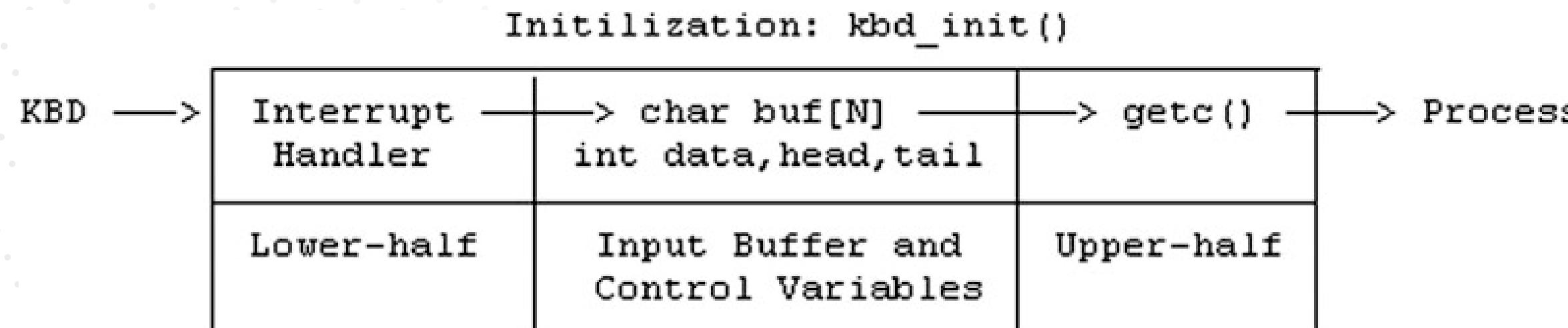
Capturar linha



Driver para o teclado

O driver para o teclado ARM Versatile, ou **KBD** (*keyboard driver*) possui três componentes:

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



Driver para o teclado

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.

Driver para o 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 desenvolvido, baseado no livro K.C. Wang. *Embedded and Real-Time Operating Systems*, lida apenas com **caracteres minúsculos**.

Parte 3 { }

Setup

Parte 1

Preparação do Ambiente de Desenvolvimento

Introdução

Parte 2

Introdução ao driver do teclado

Atividades

Parte 3

Rotinas lock e unlock

Parte 4

Driver do teclado

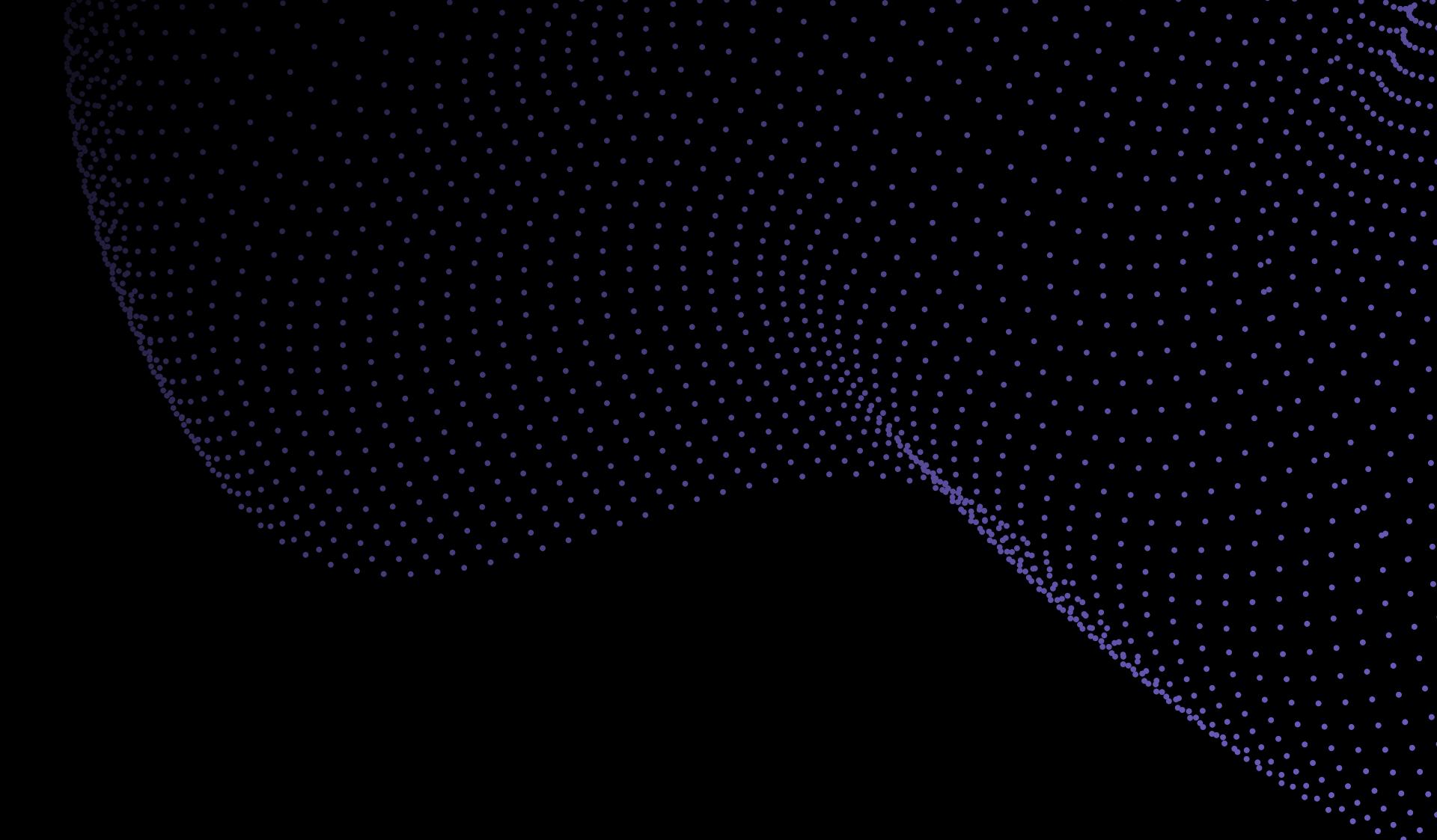
Parte 5

Modificação do programa principal

Desafio

Parte 6

Capturar linha



Atividade 1: Modificações em *ts.s*

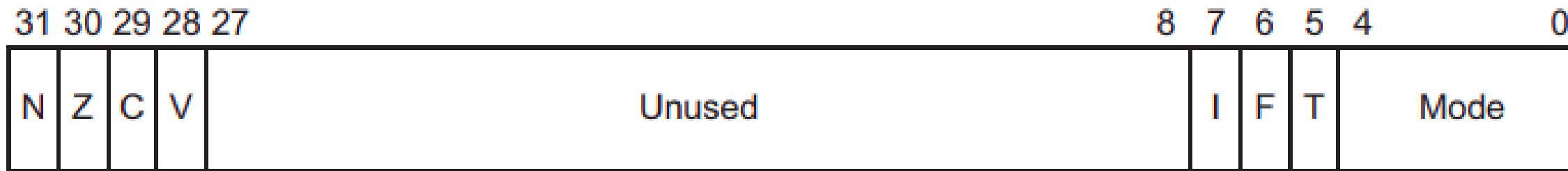
Nessa atividade espera-se criar rotinas para habilitar e desabilitar interrupções IRQ em *ts.s* e também preparar as tabelas de conversão de *scan codes* a serem usadas pelo *driver*.

Durante a recuperação de um caractere do *buffer*, será necessário desabilitar interrupções IRQ para **impedir novas modificações no buffer**.

Importante: 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*.

Atividade 1: Modificações em *ts.s*

Realiza-se a implementação de rotinas de ***lock*** e ***unlock*** em assembly para desabilitar e habilitar interrupções IRQ. A função ***lock*** deve escrever 1 no bit **I** do **CSPR**, enquanto a função ***unlock*** deve escrever 0 no bit **I** do **CSPR**.



Atividade 1: Modificações em *ts.s*

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
```

Atividade 1: Modificações em *ts.s*

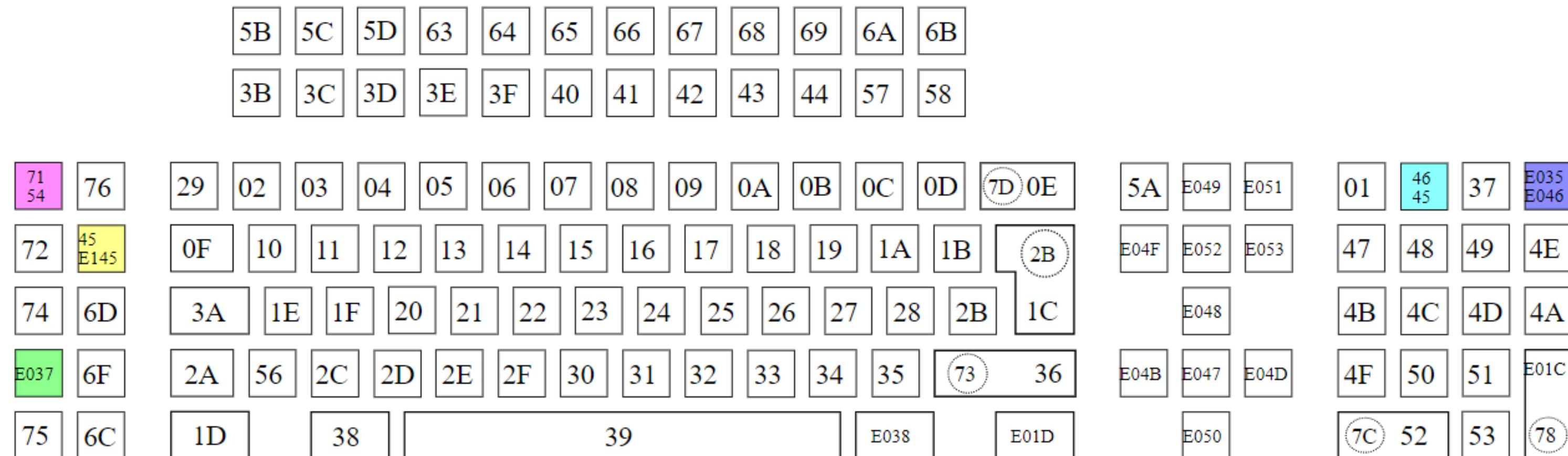
Antes de escrever o *driver*, é necessário fornecer uma tabela para conversão entre os *scan codes* fornecidos pelo teclado e o valor ASCII dos caracteres.

ltab[] - Versatile

```
2 char ltab[] = {
3     0, 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,
6     0, 'c', 'x', 'd', 'e', '4', '3', 0, 0, ' ', 'v', 'f', 't',
7     'r', '5', 0,
8     0, 'n', 'b', 'h', 'g', 'y', '6', 0, 0, 0, 'm', 'j', 'u',
9     '7', '8', 0,
10    0, ' ', 'k', 'i', 'o', '0', '9', 0, 0, 0, '/', '1', 'j',
11    'p', '–', 0,
12    0, 0, '\n', 0, '[', '=', 0, 0, 0, 0, '\r', ']', 0,
13    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
14    0, 0, 0
15};
```

Atividade 1: Modificações em *ts.s*

Exemplo de mapeamento - IBM 1397000 Keyboard



Parte 4 { }

Setup

Parte 1

Preparação do Ambiente de Desenvolvimento

Introdução

Parte 2

Introdução ao driver do teclado

Atividades

Parte 4

Driver do teclado

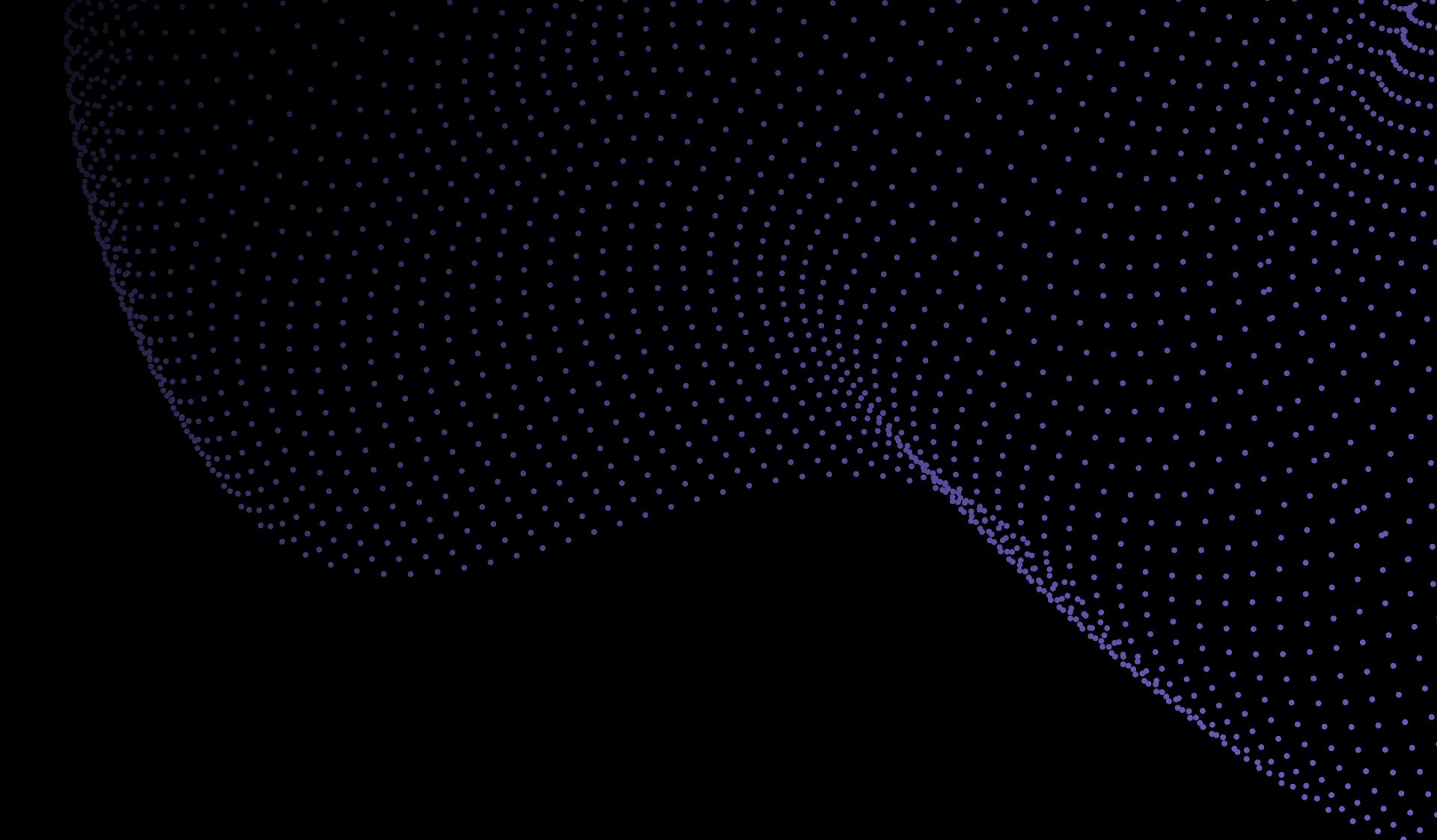
Desafio

Parte 5

Modificação do programa principal

Parte 6

Capturar linha



Atividade 2: Driver kbd.c

Nessa atividade espera-se criar duas rotinas essenciais para o *driver* do teclado - *kbd_init* e *kbd_handler*. O *kbd_handler* será utilizado para imprimir o caractere no terminal.

Analizando os registradores, o registrador de controle do teclado está localizado no endereço 0x14. Para **habilitar o teclado** deve-se **ativar o bit 2** desse registrador e para **configurar seu uso como dispositivo de Input** deve-se **ativar o bit 4**.

No início do arquivo, **importa-se as tabelas** definidas em *keymap2* e **declara-se 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
```

| Offset | Register | Bits Assignment |
|--------|-----------|-------------------------------|
| ----- | ----- | ----- |
| 0x00 | Control | bit 5=0 (AT) 4=IntEn 2=Enable |
| 0x04 | Status | bit 4=RXF 3=RXBUSY |
| 0x08 | Data | input scan code |
| 0x0C | ClkDiv | (a value between 0-15) |
| 0x10 | IntStatus | bit 0=RX interrupt |
| ----- | ----- | ----- |

Atividade 2: *Driver kbd.c*

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.

Atividade 2: Driver kbd.c

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

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

Atividade 2: Driver kbd.c

Observação 1: 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 2: logo após a liberação da tecla é gerada uma interrupção repetindo o caractere, no *interrupt handler* vamos ignorar interrupções mantendo uma variável global *release*.

Atividade 2: Driver kbd.c

```
1 int kbd_init()
2 {
3     KBD *kp = &kbd;
4     kp->base = (char *)0x10006000;
5     *(kp->base + KCNTL) = 0x10;      // bit4=ativar teclado bit0=ativar
6         interrupcao
7     *(kp->base + KCLK) = 8;
8     kp->head = kp->tail = 0;          // buffer circular char buf[128]
9     kp->data = 0; kp->room = 128;
10    release = 0;
11
12 void kbd_handler()
13 {
14     u8 scode, c;
15     KBD *kp = &kbd;
16
17     color = YELLOW;
18
19     scode = *(kp->base + KDATA);    // pega o scan code do caractere
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     // mapeia scan code para ASCII - minusculos
32     c = ltab[scode];
33
34     kp->buf [kp->head++] = c;
35     kp->head %= 128;
36     kp->data++; kp->room--;
37 }
```

Parte 5 { }

Setup

Parte 1

Preparação do Ambiente de Desenvolvimento

Introdução

Parte 2

Introdução ao driver do teclado

Atividades

Parte 4

Driver do teclado

Parte 3

Rotinas lock e unlock

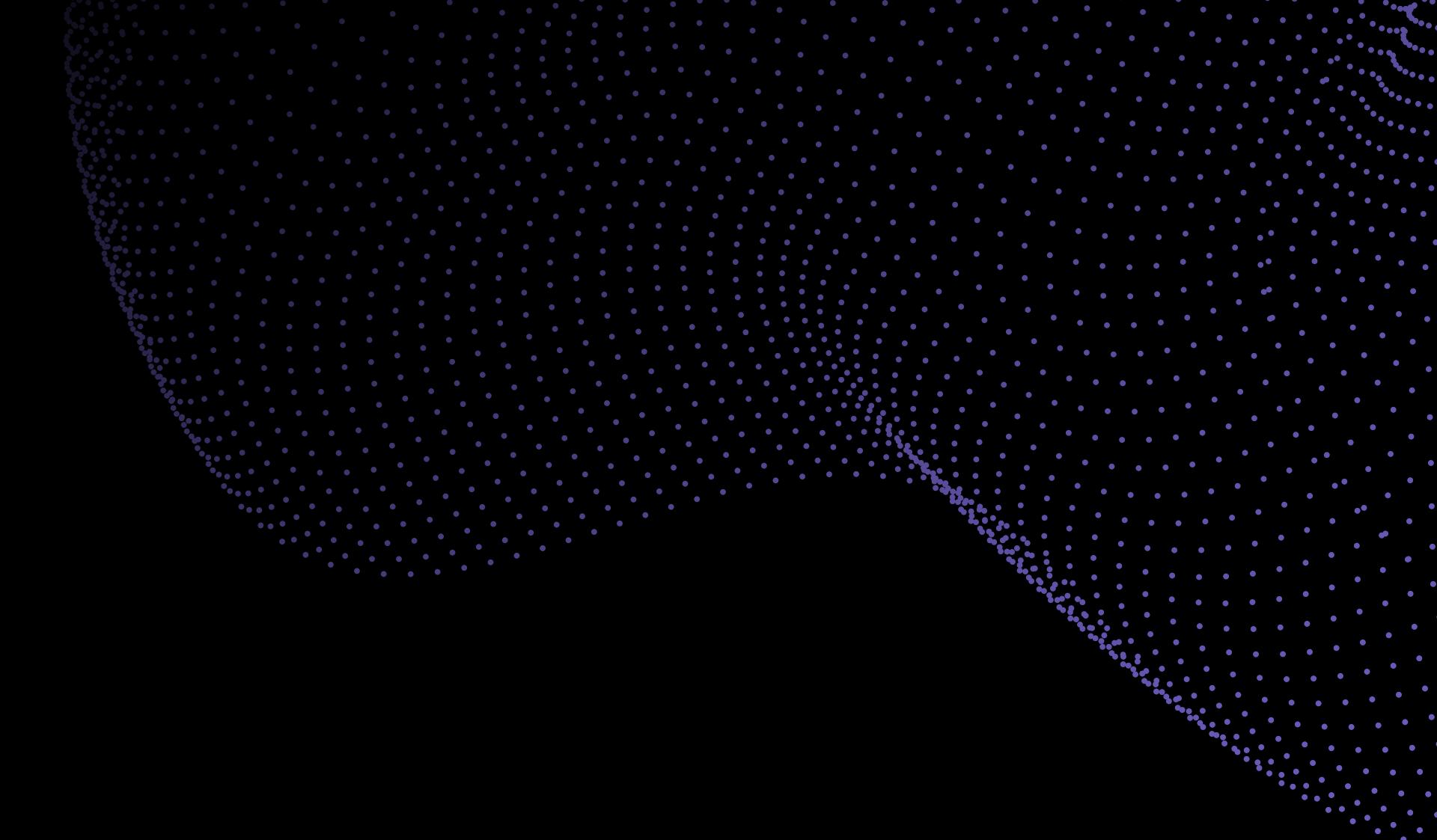
Parte 5

Modificação do programa principal

Desafio

Parte 6

Capturar linha



Atividade 3: Programa principal

Nessa atividade espera-se modificação no *handler* de interrupções IRQ e o programa principal para coletar dados do teclado.

Primeiramente, modifica-se o *handler* de interrupções IRQ para identificar interrupções de teclado, diferenciado-as das interrupções de *timer*. Essa verificação é realizada em **duas etapas** analisando os registradores:

- VIC_STATUS - no bit 31
- SIC_STATUS - no bit 3

Atividade 3: Programa principal

```
1 void IRQ_chandler()
2 {
3     int vicstatus = *(VIC_BASE + VIC_STATUS); // ler VIC status
4     int sicstatus = *(SIC_BASE + VIC_STATUS); // ler SIC status
5
6     if (vicstatus & (1<<4)){                // timer0 no bit4 do
7         VIC
8         timer_handler(0);
9     }
10
11    if (vicstatus & (1<<31)){      // VIC bit31 = SIC interrupcao
12        if (sicstatus & (1<<3)){ // KBD no bit3 do SIC
13            kbd_handler();
14        }
15    }
}
```

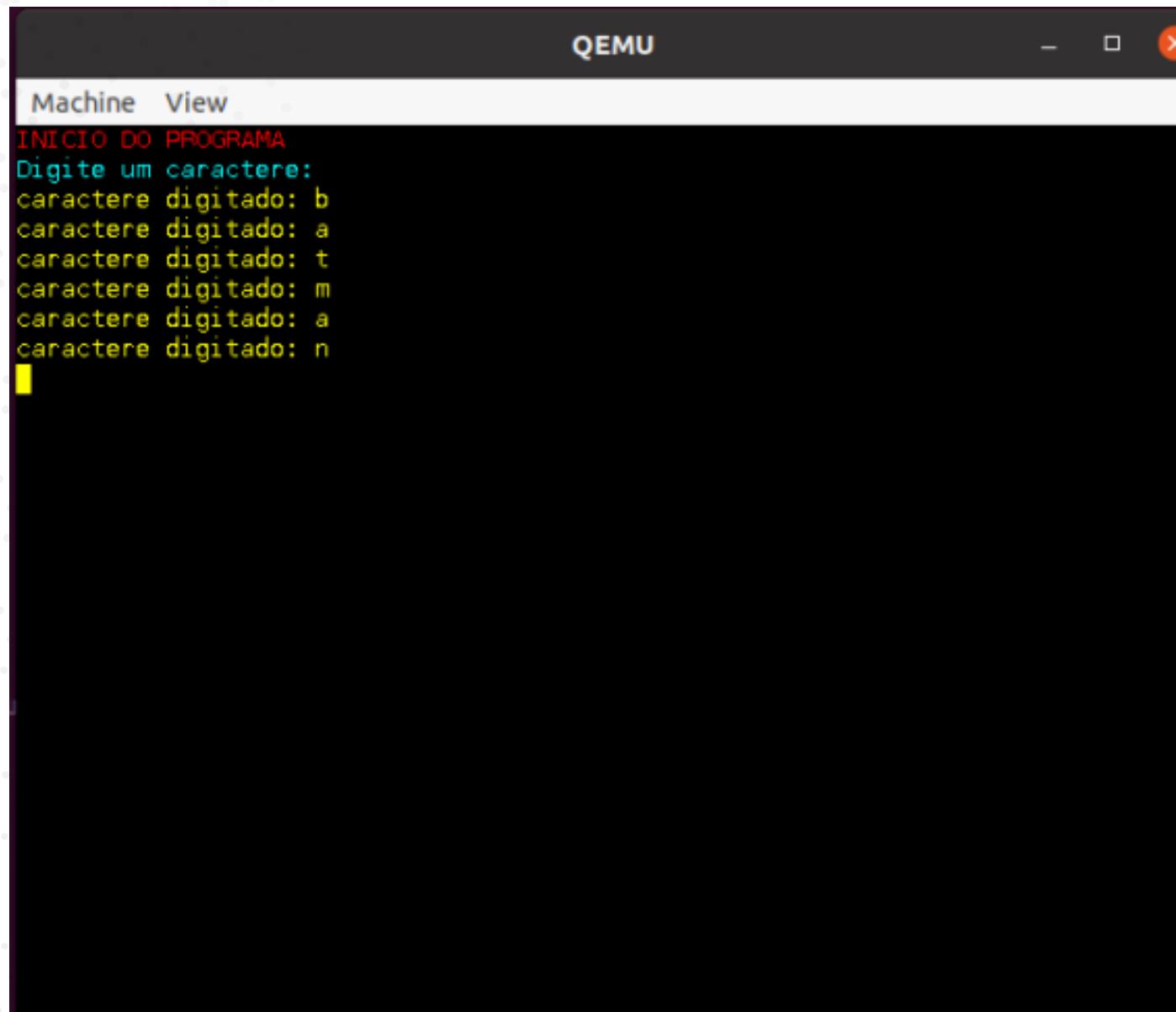
Atividade 3: Programa principal

Na rotina principal, deve-se 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);           // ativar
9     interrupcao de timer
10    *(VIC_BASE + VIC_INTENABLE) |= (1<<31);          // ativar
11    interrupcao do teclado
12    *(SIC_BASE + SIC_ENSET) |= (1<<3);               // ativar
13    interrupcao do teclado
14
15
16    color = CYAN;
17    kputs("INICIO DO PROGRAMA\n");
18    timer_init();
19    timer_start(0);
20
21    while(1);
22 }
```

Atividade 3: Programa principal

Com o *handler* modificado, é possível capturar interrupções de teclado durante o *loop* do programa principal.



Parte 6 { }

Setup

Parte 1

Preparação do Ambiente de Desenvolvimento

Introdução

Parte 2

Introdução ao driver do teclado

Atividades

Parte 4

Driver do teclado

Parte 3

Rotinas lock e unlock

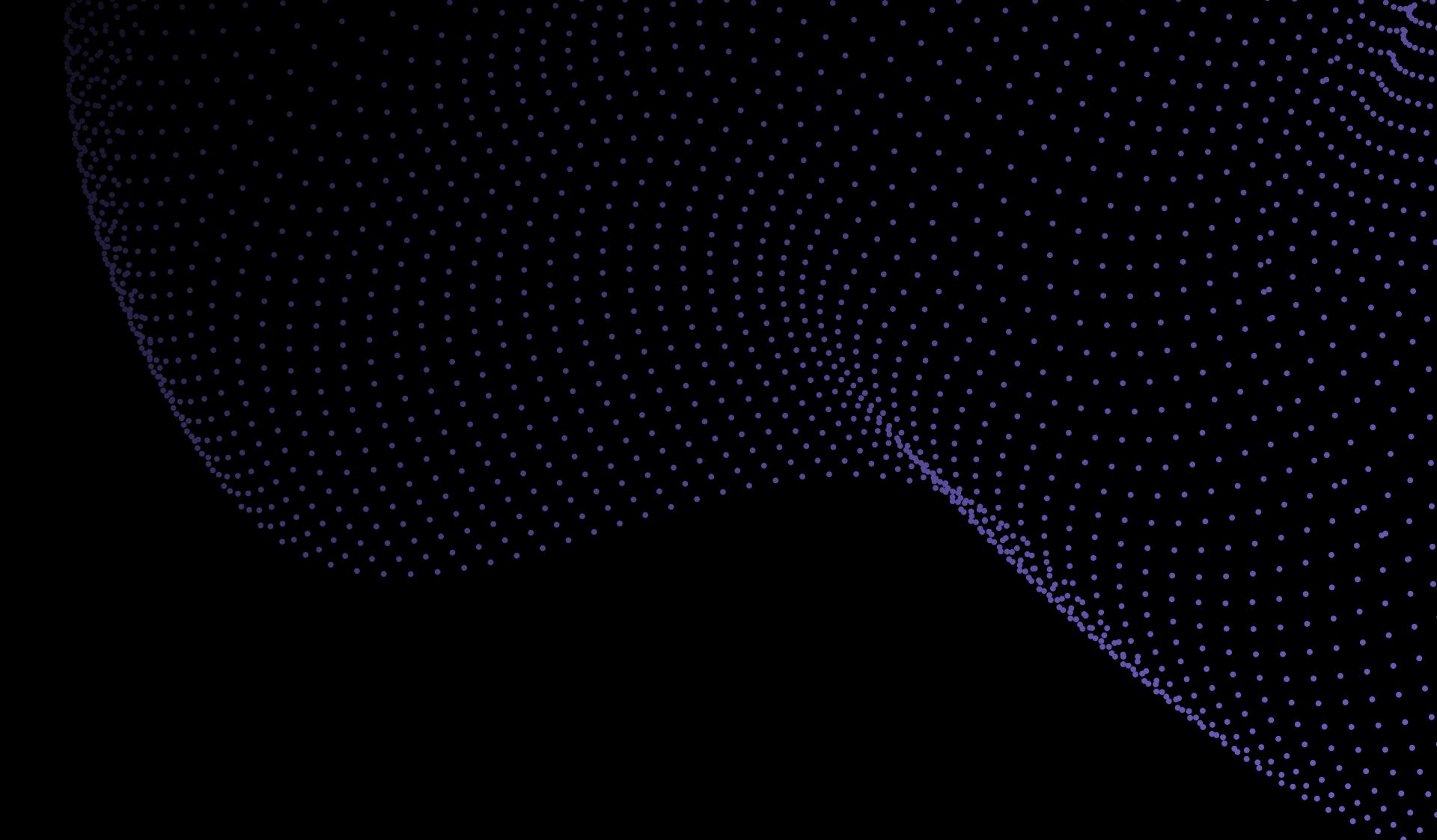
Parte 5

Modificação do programa principal

Desafio

Parte 6

Capturar linha



Desafio: Capturar linha

No desafio espera-se modificação no *driver* e no programa principal para capturar uma linha digitada e não apenas um caractere.

São desenvolvidas duas novas funções para o *driver*:

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

Observação: Atenção com a utilização **kgetc** para evitar condições de corrida no acesso ao *buffer*. Temos a desabilitação de interrupções - Atividade 1 - e o uso do MUTEX [*kp->data*] que mantém o *driver* em *busy wait* caso necessário.

Desafio: Capturar linha

```
1 int kgetc()
2 {
3     char c;
4     KBD *kp = &kbd;
5
6     unlock();                                // habilita IRQ em caso de
7     estar desabilitado
8     while(kp->data == 0);                  // BUSY wait enquanto kp->
9     data eh 0
10
11    lock();                                 // desabilita IRQ
12    c = kp->buf[kp->tail++];
13    kp->tail %= 128;                         // Regiao Critica
14    kp->data--; kp->room++;
15    unlock();                                // habilita IRQ
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 }
```

Desafio: Capturar linha

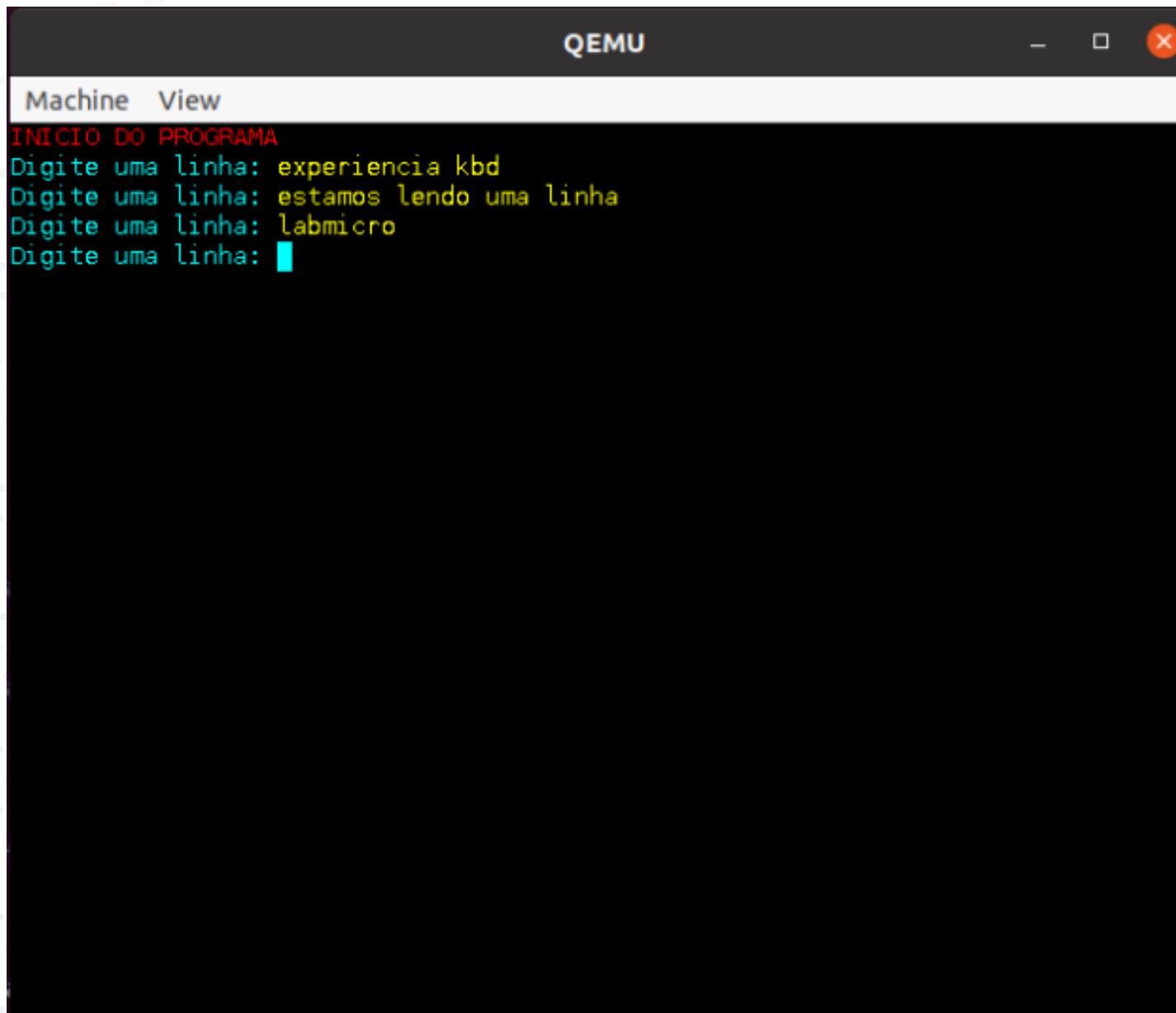
Modifica-se também o **kbd_handler** para deixar de imprimir o caractere logo após a interrupção. A impressão será realizada no *loop* do **kgets** como visto no código anterior.

Para o programa principal basta modificar o *loop* onde aguarda-se a entrada de caractere eativamente pede-se por 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      kgetc(); kputs("\n");  
6  }
```

Desafio: Capturar linha

Após as modificações espera-se que a saída se comporte como mostrado a seguir:



The screenshot shows a terminal window titled "QEMU" with a black background and white text. The window has a title bar with "Machine View" and a close button. The terminal displays the following text:
INICIO DO PROGRAMA
Digite uma linha: experiencia kbd
Digite uma linha: estamos lendo uma linha
Digite uma linha: labmicro
Digite uma linha: █

Obrigado!