Laboratório de Microprocessadores
Projeto

Prof. Dr. Jorge Kinoshita

Bancada B6

Andrei dos Santos - 10333268

Gabriel Kenji Godoy Shimanuki 10336719

Gabriel Pereira de Carvalho - 11257668

3 de agosto de 2022

Sumário

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.

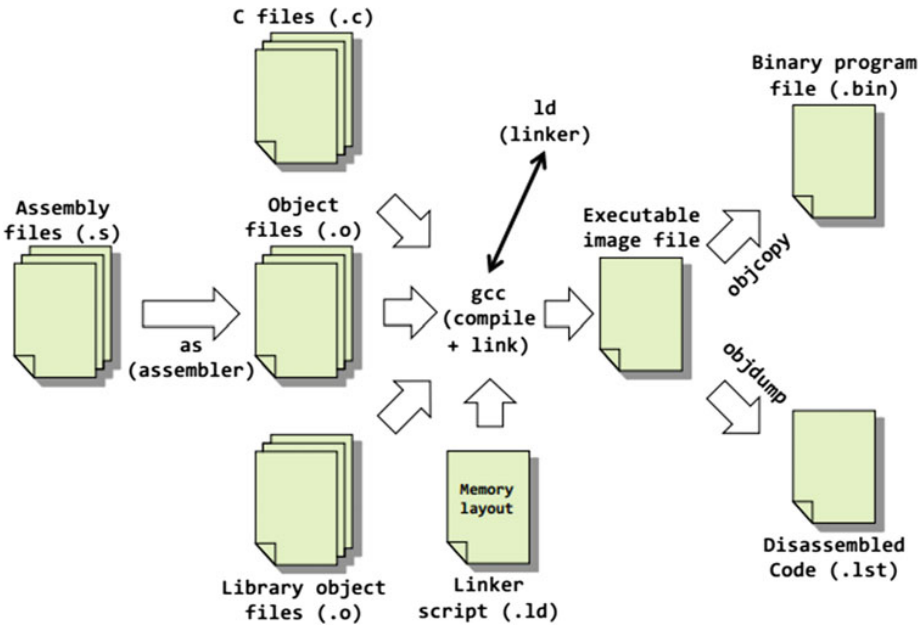

Figura 1: Componentes de uma *toolchain*

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.

```
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.

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

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

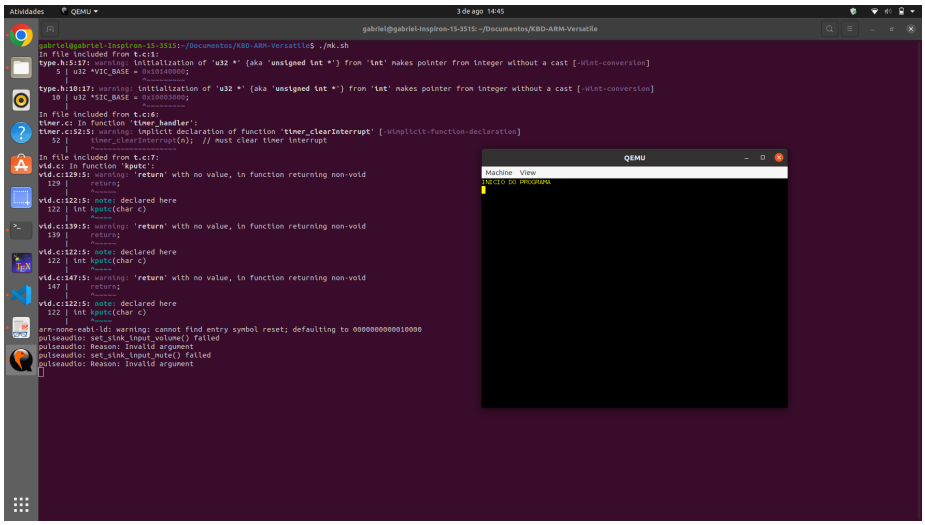

Figura 2: Executando código do repositório base

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.

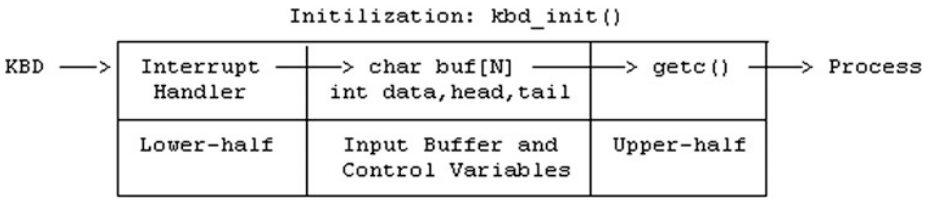

Figura 3: Componentes do *driver* do teclado

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`.

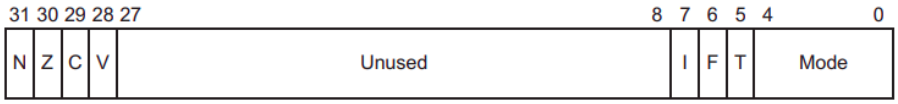

Figura 4: Localização do bit `I` no `CPSR`

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

```
.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
```

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`.

```
//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,
```

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

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 ainda livre no *buffer*.
4. variável `room` com espaço já ocupado no *buffer*.

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

volatile KBD kbd;
```

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`.

```
int kbd_init()
{
  KBD *kp = &kbd;
  kp->base = (char *)0x10006000;
  *(kp->base + KCNTL) = 0x10;    // bit4=Enable bit0=INT on
  *(kp->base + KCLK)  = 8;       // ARM manual says clock=8
  kp->head = kp->tail = 0;       // circular buffer char buf[128]
  kp->data = 0; kp->room = 128;

}

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

  color = YELLOW;

  scode = *(kp->base + KDATA);   // get scan code of this interrpt

  if (scode == 0xF0){  // ignora liberacao de tecla
     return;
  }

  // map scode to ASCII in lowercase
  c = ltab[scode];

  kputs("caractere digitado: "); kputc(c); kputs("\n");

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

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, diferenciado-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
       VIC
7        timer_handler(0);
8    }
9
10   if (vicstatus & (1<<31)){     // VIC bit31 = SIC interrupts
11      if (sicstatus & (1<<3)){   // KBD at bit3 on SIC
12           kbd_handler();
13      }
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
       timer0 at bit4
9     *(VIC_BASE + VIC_INTENABLE)  |= (1<<31);         // VIC rount
       bit31
10    *(SIC_BASE + SIC_ENSET)      |= (1<<3);         // SIC bit3 = KBD
       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`.

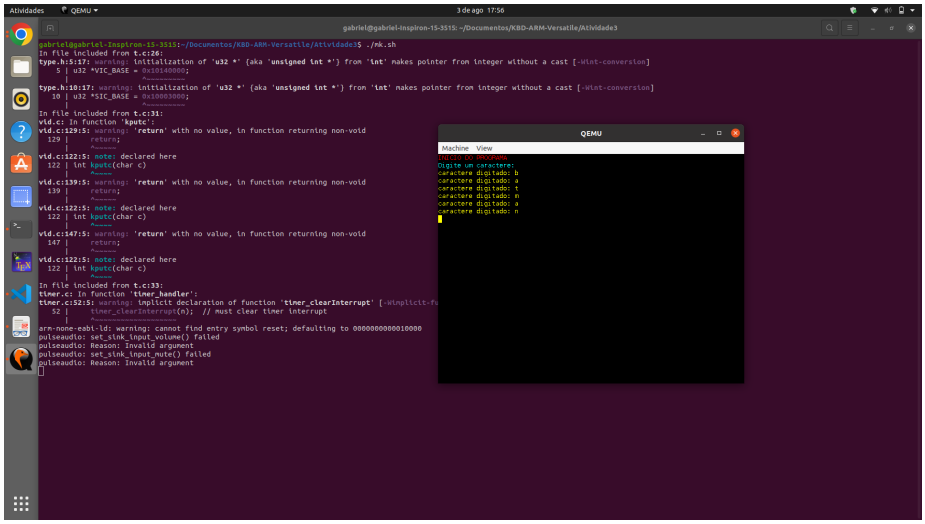

Figura 5: Execução do projeto

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.

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

  unlock();                          // unmask IRQ in case it was
    masked out
  while(kp->data == 0);              // BUSY wait while kp->data is
     0

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


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

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 aguardavamos a entrada de caracter e ativamente pedir por uma nova linha a cada iteração com a função `kgets`.

```
   while(1){
      color = CYAN;
      kputs("Digite uma linha: ");
      kputs(kgets(texto));
      kgetc(); kputs("\n");
   }
```

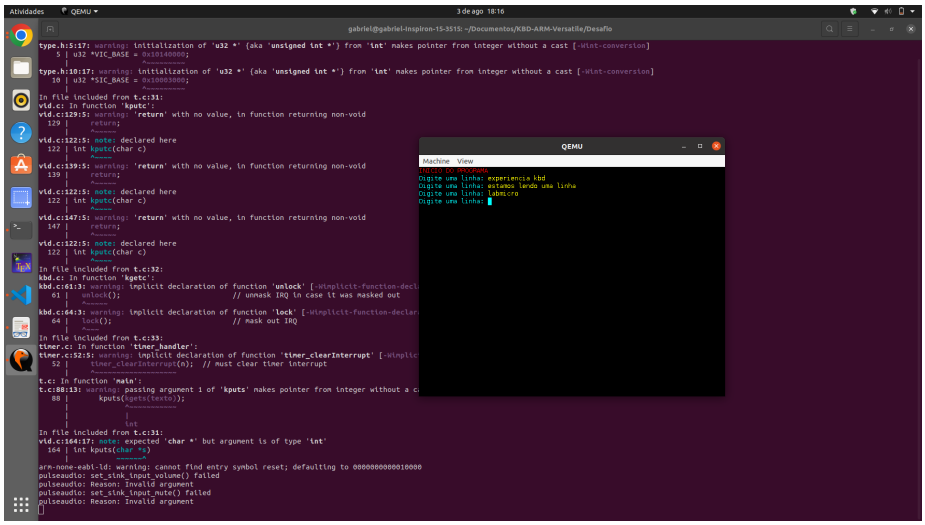

Figura 6: Execução do desafio

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).