ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DISCIPLINA: LABORATÓRIO DE PROCESSADORES- PCS3732
1° QUADRIMESTRE/2021

Prova 2

12 de Agosto de 2021

GRUPO 10

Arthur Pires da Fonseca	NUSP: 10773096
Bruno José Móvio	NUSP: 10336852
Iago Soriano Roque Monteiro	NUSP: 8572921

Sumário

Tarefa Proposta

Criar interrupções disparadas pela UART e imprimir o respectivo caractere na tela. Em paralelo, configurar um timer que irá disparar interrupções que devem imprimir um caractere `#` na tela.

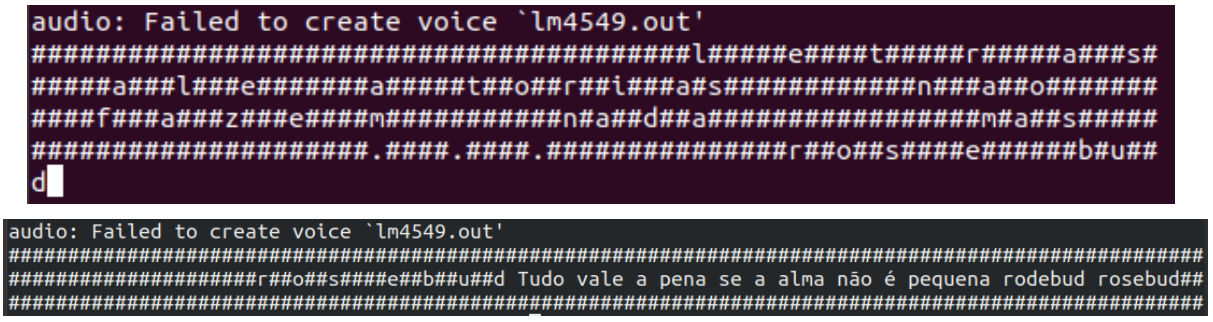
Desafio: criar lógica para interromper as interrupções de timer caso uma senha pré determinada seja escrita pela UART.

A tarefa proposta pelo grupo é imprimir no terminal as teclas que são digitadas no teclado, através do handler de interrupção.

Para tornar a tarefa mais interessante, vamos observar no console o resultado de duas interrupções se intercalando: de UART e de timer.

Conseguiremos observar o caractere “#” sendo impresso a cada interrupção de timer, intercalado pelos caracteres digitados no teclado.

Adicionalmente, implementamos uma máquina de estados que reconhece uma palavra e encerra a execução dos prints de # a cada interrupção de timer assim que a palavra inteira for digitada.



Saída do terminal, intercalando “#”s e os caracteres digitados no teclado, em duas situações diferentes

Possibilidades geradas com essa funcionalidade incluem:

- Editor de texto
- Desvio para um certo modo (ex: entrar na BIOS do computador)
- Reinício do computador: “Magic SysRq Key” + REISUB”[3]

Implementação

O projeto foi desenvolvido usando o GitHub, seu código fonte está em:

https://github.com/APF2000/LabProc/tree/uart

Os arquivos principais envolvidos na solução são o linker, em *test.ld*, o vetor de interrupções e os handlers de interrupção, em *vectors.s*, e um código em C com um handler de interrupções mais funções auxiliares, em *test.c*.

O primeiro passo é implementar o que se faz no site do Balau[4], para permitir interrupções de UART:

```
vectors_start:
 LDR PC, reset_handler_addr
 LDR PC, undef_handler_addr
 LDR PC, swi_handler_addr
 LDR PC, prefetch_abort_handler_addr
 LDR PC, data_abort_handler_addr
 B .
 LDR PC, irq_handler_addr
 LDR PC, fiq_handler_addr

reset_handler_addr: .word reset_handler
undef_handler_addr: .word undef_handler
swi_handler_addr: .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr: .word data_abort_handler
irq_handler_addr: .word irq_handler_entry
fiq_handler_addr: .word fiq_handler

vectors_end:
```

Arquivo "vectors.s", conforme especificado pelo Balau

Em seguida, vamos criar uma função que faz o setup do módulo de timer, e chamá-lo no reset_handler, que é a primeira função chamada ao se fazer o reset do sistema.

Esse handler faz a cópia do vetor de interrupção para uma nova posição de memória, além de configurar o stack pointer do modo de exceção.

O que fizemos é o código do texto do Balau, com a modificação de **inicializar o timer**:

```
reset_handler:

 /* set Supervisor stack */
 LDR sp, =stack_top

 /* copy vector table to address 0 */
```

```
BL copy_vectors

/* get Program Status Register */
MRS r0, cpsr

/* go in IRQ mode */
BIC r1, r0, #0x1F
ORR r1, r1, #0x12
MSR cpsr, r1

/* set IRQ stack */
LDR sp, =irq_stack_top

/* Enable IRQs */
BIC r0, r0, #0x80

/* go back in Supervisor mode */
MSR cpsr, r0

bl timer_init

BL main
B .

.end
```

Arquivo “vectors.s”

A função de inicializar o timer é a mesma de experiências anteriores:

```
timer_init:                                  @ configurar timer
   LDR r0, INTEN
   LDR r1,=0x10                              @ bit 4 for timer 0 interrupt enable
   STR r1,[r0]

   LDR r0, TIMER0C
   LDR r1, [r0]
   MOV r1, #0xA0                             @ enable timer module
   STR r1, [r0]

   LDR r0, TIMER0V
   LDR r1, =0xfff                            @ setting timer value
   STR r1,[r0]

   mrs r0, cpsr
   bic r0,r0,#0x80
   msr cpsr_c,r0                             @ enabling interrupts in the cpsr

   mov pc, lr
```

Essa função habilita as interrupções de timer e seta o valor do timer para um valor maior: 0xfff ao invés do 0xff anterior, que faz as interrupções acontecerem num ritmo menos acelerado.

A main chamada pelo reset_handler é implementada em C, que habilita interrupções de UART e fica num loop infinito sem fazer nada. Assim, tudo o que se vê no console são os prints produzidos nos handlers das duas interrupções que estamos produzindo.

```
int main(void) {

 /* enable UART0 IRQ */
 VIC_INTENABLE = 1<<12;

 /* enable RXIM interrupt */
 UART0_IMSC = 1<<4;

 for(;;);
}
```

O entry-point para ambas as interrupções de timer e de UART é o mesmo: a função *irq_handler_entry* chamada no vetor de interrupções.

Essa função é dividida em três partes:

- Primeiro, algumas configurações anteriores à recuperação de registradores é feita: o endereço para onde irão os registradores do processo que foi interrompido são determinados pela quantidade de processos esperando para voltarem de suas interrupções naquele momento (label “qtde_subprocs”) e pelo endereço base onde os valores dos registradores podem ficar armazenados sem sobrescrever instruções já presentes na memória
- Depois, decide-se qual é o tipo de interrupção que motivou aquela chamada (na seção *trata_interrupção*)
- Então, desvia-se para o tratamento da interrupção adequada e se recupera os registradores em seguida

Para interromper as impressões de “#” após determinada senha ser digitada na UART é implementada no código em C uma função irq_handler, que é disparada pela UART após toda interrupção. Nela é identificado o caractere digitado e impresso na tela, em seguida é feita uma validação com switch / case para identificar se o caractere digitado pertence à senha pré definida ou não.

No nosso caso de teste a palavra-chave utilizada foi **rosebud**.

```
void __attribute__((interrupt)) irq_handler(){
 /* echo the received character + 1 */

          char c = UART0_DR;
          char st[2];
          st[0] = c;
          st[1] = '\0';
          print_uart0(st);
 switch(c)
 {
           case 0x72: // letra r em hexadecimal ascii
                    prox_passo(0);
```

```
                break;
        case 0x6f: // letra o em hexadecimal ascii
                prox_passo(1);
                break;
        case 0x73: // letra s em hexadecimal ascii
                prox_passo(2);
                break;
        case 0x65: // letra e em hexadecimal ascii
                prox_passo(3);
                break;
        case 0x62: // letra b em hexadecimal ascii
                prox_passo(4);
                break;
        case 0x75: // letra u em hexadecimal ascii
                prox_passo(5);
                break;
        case 0x64: // letra d em hexadecimal ascii
                prox_passo(6);
                break;
        default:
                state = 0; // reseta a maquina de estados
                return;
 }
}
```

A fim de realizar a identificação da escrita da palavra, criou-se uma variável auxiliar de estado, que deve armazenar um valor de 0 a 6 indicando qual a próxima letra que deve ser digitada.

Caso siga digitando na ordem correta, o switch / case irá selecionar a chamada da função **prox_passo** com o target correto, permitindo o incremento da variável state.

Caso o usuário tecle mais de uma vez um caractere da senha (no nosso exemplo onde a senha é **rosebud** imagina-se uma situação onde o usuário digita **ross**), a segunda ocorrência não irá alterar o estado, permitindo que o usuário continue digitando.

Quando o valor de estado alcança o valor de 6, altera-se a variável **can_print** para o valor inverso do atual. Dessa forma, a variável funciona como um interruptor que libera ou não a impressão do caractere “#”.

A seguir é possível visualizar a implementação da função **prox_passo** e da função **print_interrupcao**, onde ocorre a verificação para permitir a impressão de “#”.

```
void prox_passo(int target) {
        if(state == target + 1) { // condição de saída de repetição
                return;
        }

        if(state == target) {
                if(state == LAST_STATE) {
                        can_print = !can_print;
                        state = 0;
                }else{
                        (state)++;
                }
```

```
        }
        else{
                (state) = 0;
        }
}
```

```
void print_interrupcao() {
        if(can_print){
                print_uart0("#");
         }
 }
```

Referências

[1] Github do Kassel Wang
https://github.com/smwikipedia/EmbeddedAndRTOSSamples

[2] Livro do Kassel Wang
https://www.amazon.com.br/Embedded-Real-Time-Operating-Systems-K-C/dp/3319846728

[3] Magic SysRq Key
https://en.wikipedia.org/wiki/Magic_SysRq_key

[4] Blog do Balau
https://balau82.wordpress.com/2012/04/15/arm926-interrupts-in-qemu/