Sistemas Embebidos – 2020/2021

# Máquina de Vendas Automática

Primeiro Trabalho de Sistemas Embebidos

Milena Mori
2016193815

Lucyanno Frota
2016116214

## 1 Introdução

O objetivo desse trabalho era implementar uma máquina de vendas automática, com a utilização da placa STM32L4R5ZI. Utilizamos uma tela LCD como display, e comunicação serial para entrada de dados. Nesse trabalho, utilizamos 5 tarefas: Uma para a máquina de estados, uma para o display, uma para capturar inputs por comunicação serial, uma para fazer um display por comunicação serial e outra para a detecção do botão.

## 2 Máquina de Estados (defaultTask)

O primeiro passo para montar a máquina de vendas, era pensar na máquina de estados. Como podemos ver na figura 1, temos seis estados: INIT, WAIT, ADD, SELECT PRODUCT (SEL), DISPENSE (DISP) e BLINK. A mudança de estados depende das variáveis coinV (coin value), selectedprod (selected product), prodV (product value), tValue (total value) e sensor.

Sendo que: coinV é o valor atual da moeda (se não tiver sido inserida uma moeda, o valor é zero), selectedprod é o produto selecionado (valor depende de linha + coluna), prodV é o valor do produto selecionado (se não tiver sido selecionado um produto, prodV = 0), tValue é o valor total (soma de todos os coinV desde que o último valor foi vendido) e sensor é um booleano que diz se o produto foi dispensado. A nossa máquina não libera troco.

### 2.1 Estados

**INIT** Nesse estado, todas as variáveis locais e todas as saídas são inicializadas a zero. O próximo estado é sempre WAIT.

**WAIT** Desse estado, ficamos à espera de alguma das variáveis de estado serem alteradas. Caso seja inserida uma moeda coinV o próximo estado é ADD. Caso contrário, temos várias outras possibilidades: se um produto tiver sido selecionado pela primeira vez (ou seja, $(selectedprod \neq 0)$ & $(prodV = 0)$), o próximo estado é SEL; se o valor total acumulado for maior ou igual ao valor do produto selecionado (i.e. $prodV \leq tValue$), o próximo estado é DISP. Se nenhuma das condições descritas forem satisfeitas, o próximo estado é WAIT.

**ADD** Nesse estado, é somado ao valor total o valor da moeda que foi inserida. Ou seja, $tValue+ = coinV$. O próximo estado é sempre WAIT.

**SEL** Nesse estado atribuímos à variável prodV o valor do produto. Para simplificar como que isso é feito, ao invés de atribuir valores aleatórios para cada produto, decidimos atribuir o valor dependendo da coluna em que o produto está, como mostra a tabela 1.

Os valores estão em cêntimos, para facilitar os cálculos, já que temos moedas desde 10 cêntimos até 2 euros. O próximo estado é sempre WAIT.

**DISP** Nesse estado, mandamos um sinal para que o produto certo seja dispensado e mudamos

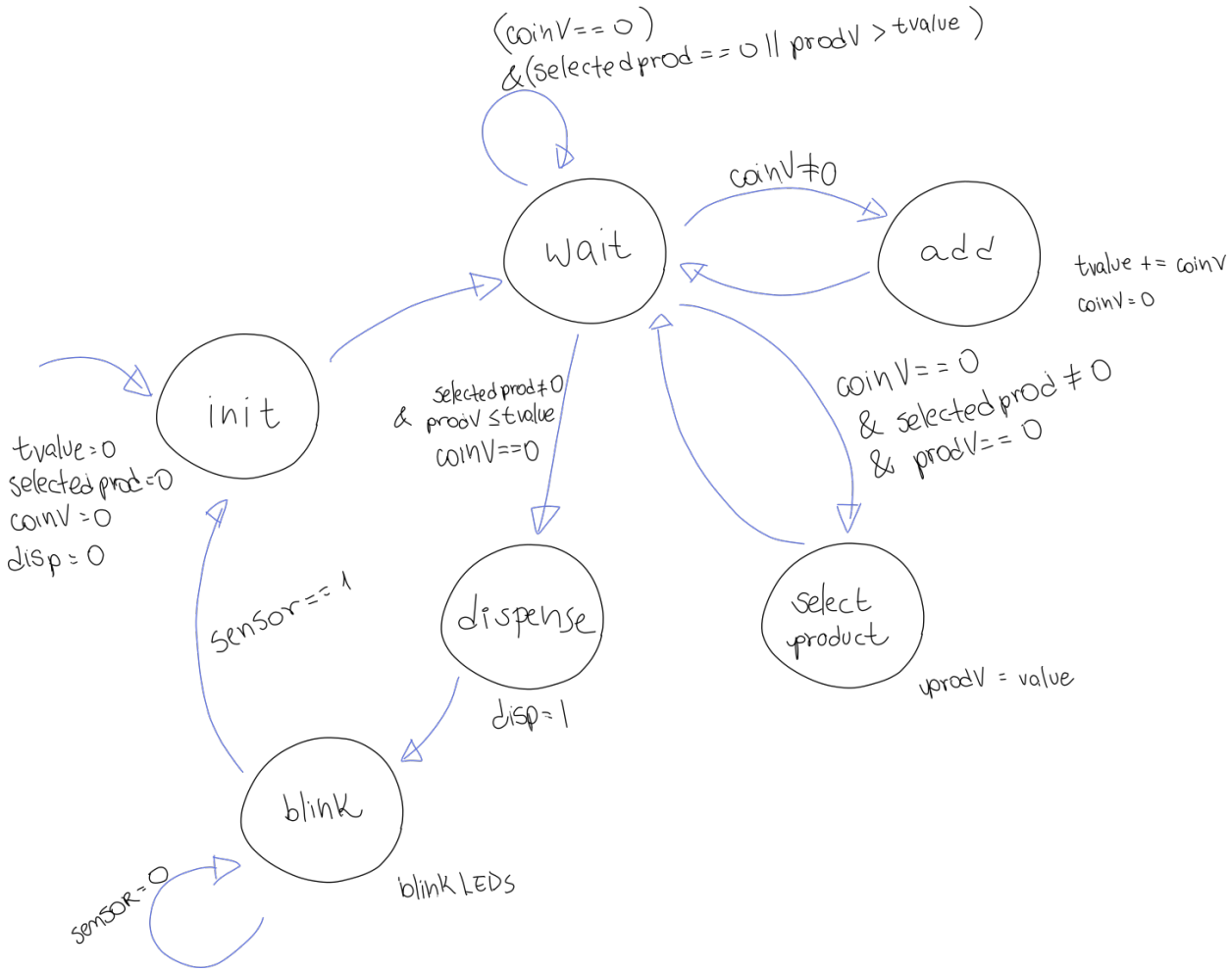


Figure 1: Máquina de estados

| Coluna | Valor |
|---|---|
| 1 | 200 |
| 2 | 250 |
| 3 | 300 |
| 4 | 250 |
| 5 | 200 |

Table 1: Valores dos produtos, dependendo da coluna

de estado. O próximo estado é sempre BLINK.

**BLINK** Nesse estado, piscamos os LEDs sequencialmente (vermelho, azul e verde, nessa ordem), enquanto não for ativado o sensor (ou seja, enquanto o sensor não for ativado, o próximo estado é o próprio estado). Quando $sensor = 1$, todos os LEDs piscam rapidamente duas vezes, simbolizando que o produto foi dispensado ao cliente. O próximo estado é então INIT.

## 3 Botão (buttonDetection)

Ao invés de fazer uma interrupção, achamos que seria melhor fazer uma tarefa para administrar o funcionamento do botão, já que ele só interfere na máquina de estado durante o estado BLINK. Nessa task, verificamos se a máquina de estados está no estado BLINK (se a variável *disp* estiver a 1) e se o botão foi pressionado. Se sim, colocamos a variável *sensor* a 1.

## 4 Input Por Comunicação Serial (SerialInput_Detect)

Para capturar os inputs por comunicação serial, utilizamos as Queues do FreeRTOS. Utilizamos um buffer para receber as informações da fila. Com a função osMessageQueueGetCount() verificamos o número de mensagens na fila, se tivermos uma mensagem ou mais, piscamos uma luz azul e utilizamos a função osMessageQueueGet() e a função strncat() para colocar a mensagem recebida no buffer. Depois de lidas todas as mensagens da fila, utilizamos a nossa função stringToCommand() para traduzir o conteúdo do nosso buffer para uma linguagem que o nosso programa entenderá e colocamos a tradução na fila para que a tarefa da máquina de estados utilize essa informação mais tarde.

A codificação dos inputs do nosso programa é feita como é apresentada na tabela 2.

| Código | Significado |
|---|---|
| $[C,v,...]$ | Moeda com valor $v$ |
| $[P,r,c]$ | Produto na linha $r$ e coluna $c$ |

Table 2: Tabela com a codificação que utilizamos para a implementação da máquina de vendas

Para que a informação seja interpretada corretamente, nós utilizamos uma estrutura chamada de Command, que está apresentada abaixo:

```
enum CType{
        coin = 0,
        product = 1,
        unknown = -1,
};
struct Command {
        enum CType type;
        int16_t value1;
        int16_t value2;
};
```

Por exemplo, se for selecionado o um produto através da mensagem [p,3,2], a informação seria passada pela função stringToCommand(), que devolveria a estrutura {type = 1, value1 = 3, value2 = 2} para ser interpretada pela máquina de estados.

Assim, podemos escrever um pseudocódigo para descrever a tarefa:

```
void SerialInput_Detect(){
    while(1){
    count = getQueueLength(Queue);
    if(count > 0){
        blinkBlue();
        for(int i = 0; i < count; i++){
            str = getMsgFromQueue(Queue);
            buffer.append(str);
        }
        struct Command a = str2Command(buffer);
        insertInQueue(Queue2, &a);
    }
    }
}
```

Em que getQueueLength() é a função osMessageQueueGetCount() e getMsgFromQueue() é osMessageQueueGet().

## 5 Display (Serial_Display)

A informação apresentada no display depende do estado da máquina de estados, se foi inserida uma moeda, e se foi selecionando um produto. Essa informação era passada para a tarefa com a utilização de uma variável de estado chamada de DispState do tipo SerialDisp:

```
enum SerialDisp{
        home = 0,
        vending = 1,
        dispensing = 2,
};
```

Essa variável era alterada dentro da máquina de estado. As opções de texto podem ser observadas na tabela 3.

| Texto | Situação |
|---|---|
| Insert a coin or select a product | Ainda não foi selecionado um produto ou inserida uma moeda |
| Price = XXX Total = YYY | Se foi selecionado um produto (XXX) ou se foi colocada uma moeda (YYY). |
| Dispensing product ZZ | Se o total for maior que do que o valor do produto. |

Table 3: Opções de texto

Essas strings são passadas por serial com a utilização da função HAL_UART_Transmit().

Essa tarefa passa para a tarefa LCD_Display as informações de estado do LCD.

## 6 LCD Display (LCD_Display)

O texto que é apresentado no LCD é o mesmo que o que é apresentado no Serial Display, como apresentado na tabela 3. Para que essa informação apareça na tela LCD, utilizamos como base a biblioteca do Deep Blue[1].

Era preciso adaptar as funções dadas por esse tutorial para o nosso contexto, utilizando como base principalmente a *datasheet* da nossa tela, já que cada modelo de LCD tem um processo de inicialização diferente.

Para fazer essa implementação, implementamos 4 estados: LCDidle, LCDhome, LCDvending e LCDdispensing. Cada um desses estados simplesmente mudava o texto que era mandado para o LCD, e depois mudavam o estado para o estado LCDIdle, que mantinha o que estava escrito no LCD, já que, a não ser que o texto fosse alterado, não era preciso mandar informação para a tela.

Sempre que fôssemos alterar o texto do LCD, tínhamos que limpar a tela, com a utilização da função LCD_Clear(), de forma que o texto fosse escrito corretamente.

## 7 Saída

Para simular a saída, já que não temos motores reais e a saída é dada digitalmente, nós separamos os dados em uma estrutura do tipo Command, que já foi descrita anteriormente. Ou seja, as informações da linha e da coluna em que o produto se encontra estão armazenadas separadamente.

A figura 2 descreve como seria feito o tratamento de dados se estivéssemos utilizando motores e multiplexers reais.

Para representar o sinal de entrada do primeiro mux, nós ligamos um LED amarelo quando é liberado algum produto (no estado DISP da máquina de estados).

## 8 Pinout

Na tabela 4, temos os pinouts. Em que os pins cuja segunda coluna está precedida por LCD correspondem a inputs da tela LCD, o LED amarelo corresponde ao LED (não acoplado à placa) que é aceso quando a máquina dispensa um produto, os outros LEDs são os LEDs (RGB) acoplados à placa.

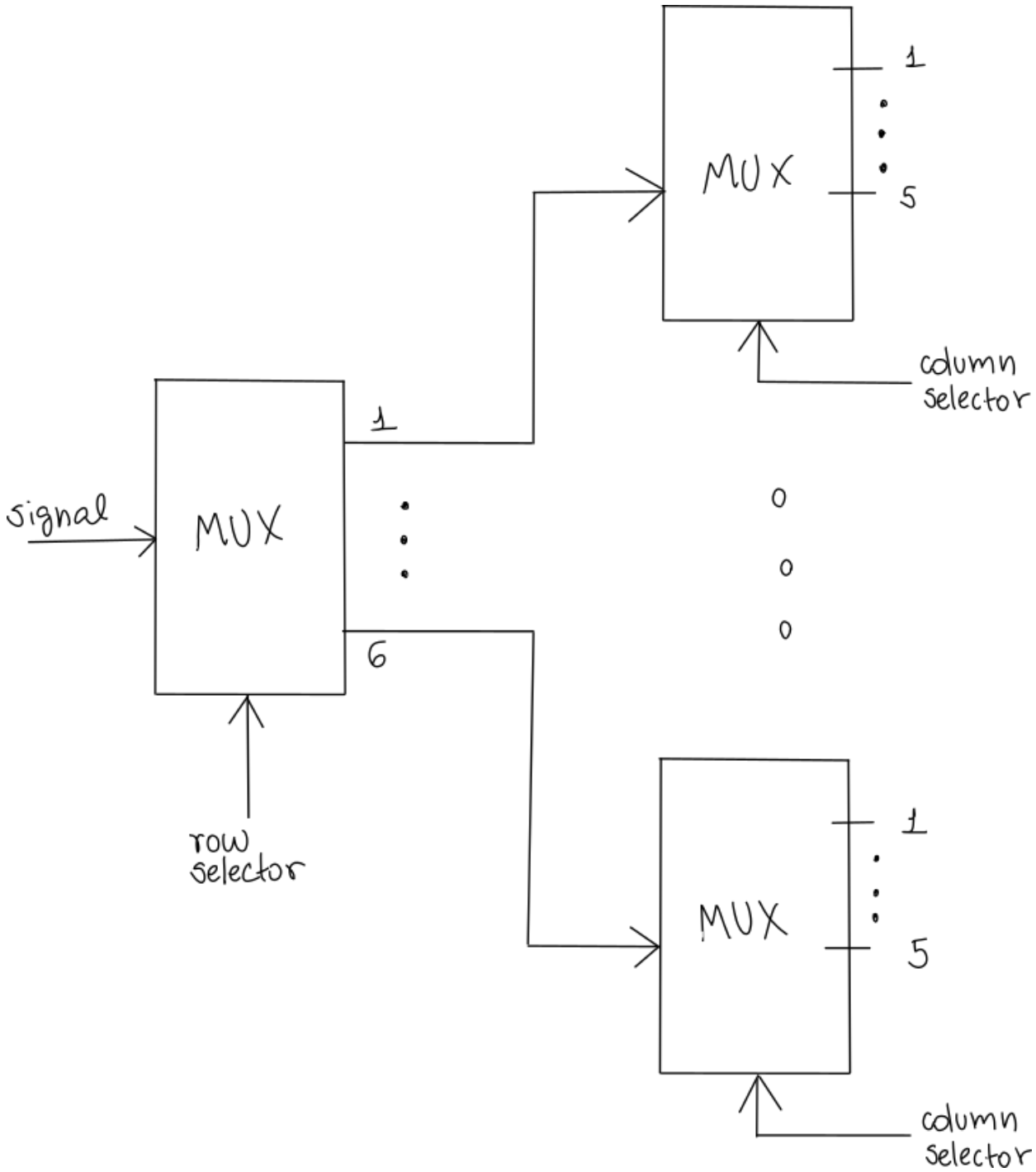


Figure 2: Descrição do que seria a saída se estivéssemos utilizando motores reais.

| Pin Number | Symbol | Nome | Function |
|---|---|---|---|
| PB2 | LCD RS | LCD_RS | Seleção de função |
| PB6 | LCD R/W | LCD_RW | Alterna entre modos read/write |
| PF2 | LCD EN | LCD_EN | Necessário para interagir com a tela |
| PF1 | LCD DB4 | LCD_DB4 | Entrada de Dados |
| PF0 | LCD DB5 | LCD_DB5 | Entrada de Dados |
| PD0 | LCD DB6 | LCD_DB6 | Entrada de Dados |
| PD1 | LCD DB7 | LCD_DB7 | Entrada de Dados |
| PD7 | LED Amarelo | YellowLed | - |
| PB7 | LED Azul | blue_Led | - |
| PC7 | LED Verde | green_Led | - |
| PB14 | LED Vermelho | red_Led | - |

Table 4: Caption

## 9 Ficheiros e Detalhes Técnicos

Os ficheiros necessários para que o projeto funcione são:

- main.c
- functions.h
- functions.c
- lcd.h
- lcd.c

Utilizamos um clock de 64MHz, 5 Tasks e 2 Queues do FreeRTOS, e ativamos o LPUART1 em modo assíncrono, sendo os seus parâmetros:

- Baud Rate: 115200 Bits/s
- Word Length: 8 Bits (including parity)
- Parity: None
- Stop Bits: 1

As funções dentro do arquivo functions são:

```
void popString(char *, char *, uint8_t);
struct Command stringToCommand(char *);
uint16_t readBuffer(enum CType *t,
↪  osMessageQueueId_t *CommandQueuHandle);
void blinkRed(uint16_t);
void blinkGreen(uint16_t);
void blinkBlue(uint16_t);
void dispLEDs(uint16_t);
void blinkAll(uint16_t);
```

As funcionalidades de cada uma desses funções podem ser encontradas na tabela 5.

Já as funções que se encontram no arquivo lcd.c são:

```
void LCD_DATA(unsigned char Data);
void LCD_CMD(unsigned char CMD);
void LCD_Init();
void LCD_Write_Char(char Data);
void LCD_Write_String(char *str);
void LCD_Clear();
void LCD_Set_Cursor(unsigned char r, unsigned
↪  char c);
void LCD_SR();
void LCD_SL();
void LCD_Write(char *l1, char *l2);
```

As funcionalidades de cada uma desses funções podem ser encontradas na tabela 6.

Como já mencionado anteriormente, não fizemos o código para manipular o LCD de raiz, mas adaptamos o de um tutorial[1]. A única função que nós criamos no arquivo lcd.c foi a função LCD_Write().

## Bibliografia

[1] D. Blue, "Stm32 lcd 16×2 tutorial & library." [Online]. Available: https://deepbluembedded.com/stm32-lcd-16x2-tutorial-library-alphanumeric-lcd-16x2-interfacing/

| Nome da Função | O que faz |
|---|---|
| blinkRed() | Mantém o LED vermelho aceso durante o tempo dado na entrada, dado em ms |
| blinkGreen() | Mantém o LED verde aceso durante o tempo dado na entrada, dado em ms |
| blinkBlue() | Mantém o LED azul aceso durante o tempo dado na entrada, dado em ms |
| dispLEDs() | Pisca em sequência os LEDs vermelho, azul e verde, nessa ordem. |
| popString() | Retira o primeiro elemento de uma string, que a função recebe como entrada |
| readBuffer() | Utilizada na máquina de estados para ler as informações colocadas na Queue, que estão em structs do tipo Command |
| stringToCommand() | Serve para "traduzir" as strings recebidas serialmente em structs Command que a máquina de estados consegue entender. |

Table 5: Funções que se encontram no arquivo functions.c

| Nome da Função | O que faz |
|---|---|
| LCD_DATA() | Manda dados para a placa |
| LCD_CMD() | Manda comandos para a placa |
| LCD_Init() | Inicializa a placa, o que é necessário para poder utilizá-la |
| LCD_Write_Char() | Escreve um char na placa |
| LCD_Write_String() | Escreve uma string na placa |
| LCD_Clear() | Limpa o texto da placa |
| LCD_Set_Cursor() | Coloca o cursor onde queremos escrever (linha, coluna) |
| LCD_SR() | Desloca todo o texto para a direita |
| LCD_SL() | Desloca todo o texto para a direita |
| LCD_Write() | Escreve a primeira string na primeira linha e a segunda string na segunda linha. |

Table 6: Funções que se encontram no arquivo lcd.c