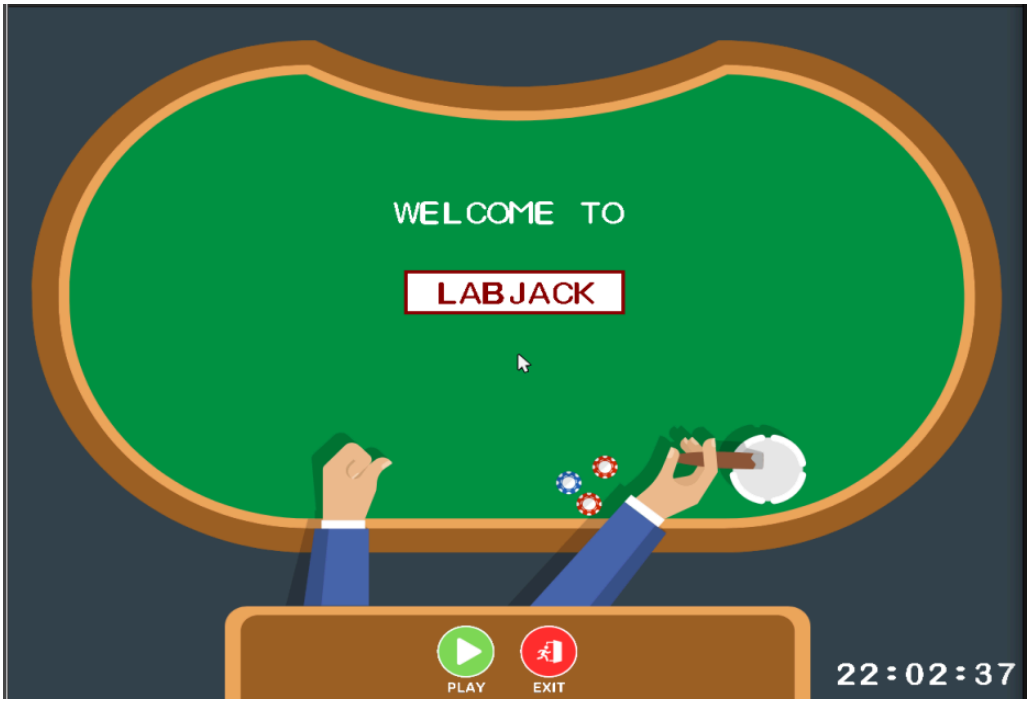
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

L.EIC018 - Laboratório de Computadores

Labjack



Uma implementação de Blackjack em C

2LEIC05 - G03
Gabriel Fragoso Lima up202206693
João Pedro Nogueira da Hora Santos up202205794
Marta Duarte Lopes da Silva up202208358
Sara de Oliveira Cortez up202205636

Índice

1. Modo de Jogo

O nosso jogo é uma implementação para minix do jogo de apostas Blackjack, ao estilo dos jogos de casino online.

O jogo é composto por uma mesa e um baralho de cartas e tem como intervenientes, o jogador (utilizador) e o dealer (lógica de computador), procedem da seguinte forma:

1. O jogador lança uma aposta.
2. O jogador (**PLAYER**) e o computador (**DEALER**) recebem duas cartas. Todas estão viradas para cima, exceto uma das cartas do dealer. As cartas numéricas valem o seu número, os Valetes, as Damas e os Reis valem cada 10 pontos. Os "Ás" podem valer 1 ou 11, consoante a necessidade. O objetivo de cada jogador é que a sua mão esteja mais perto de 21 pontos do que a do o dealer, mas sem exceder esse valor. Para isso, pode escolher entre:
    a. pedir mais cartas do baralho (**HIT**), o que aumenta o seu risco de atingir os 21 pontos,
    b. ficar com as cartas atuais e passar a vez ao dealer (**STAND**),
    c. duplicar a aposta atual, o que implica receber uma carta, passando de seguida a vez ao dealer (**DOUBLE**),
    d. render-se, ficando com metade da aposta inicial (**FORFEIT**).

    O jogador pode pedir cartas (**HIT**) quantas vezes quiser, desde que não atinja os 21 pontos antes de fazer uma das outras escolhas. As restantes jogadas são finais.
3. Depois disso, o dealer revela a carta que estava virada para baixo, e tira mais cartas do baralho até atingir os 17 ou mais pontos.
4. Se um dos intervenientes (dealer/jogador) exceder os 21, ganha o outro. Se isso não acontecer, ou seja, se a mão do jogador for mais alta que a do dealer, o jogador ganha. No caso de um empate, o jogador fica com o dinheiro que apostou.

Assim a lógica de jogo procede em cinco estados:

1. **MAIN_MENU** - O menu inicial contém opções de início de uma ronda (**START**) e de saída (**EXIT**) da aplicação.
2. **GAME_BET** - O menu de apostas, que contém o campo para input da aposta, que é um número entre 1 e o saldo do jogador.
3. **GAME_PLAY** - Este é o estado onde o jogador escolhe, após a inserção do valor da aposta, a sua próxima ação entre **HIT**, **STAND**, **DOUBLE** ou **FORFEIT**.
4. **GAME_DEALER_TURN** Representa a jogada do dealer, antes de transitar para o estado final.

5. **GAME_OVER**. O estado **GAME_OVER** concretiza-se num dos seguintes 4 subestados:
    a. **PLAYER_WIN -** O jogador ganha a mesa.
    b. **PLAYER_LOSE** - O jogador perde o dinheiro que apostou.
    c. **PLAYER_DRAW** - O jogador empata com o dealer, recebendo de volta a sua aposta.
    d. **PLAYER_SURRENDER** - O jogador desiste da ronda e recebe de volta metade da sua aposta.

2. Estado do Jogo

Toda a lógica de jogo descrita na secção anterior foi implementada. A tabela abaixo sumariza os drivers implementados e o seu papel na implementação.

2.1. Funcionalidades implementadas

- **Lógica de Jogo**

Lógica de estados, simulando Programação Orientada a Objetos através da utilização de c-structs.

- **Triple buffering com page flipping**

Na função **`vg_map_memory`** cria três frame buffers e mapeia-os para a memória física, de modo a melhorar a performance do jogo. Isto concretiza-se da seguinte forma:

    - **`gph.frame_buffer`** é um array consistindo de três apontadores, os 3 buffers do triple buffering.
    - O tamanho de cada frame buffer é determinado pela resolução do ecrã e pelo modo de indexação da cor.
    - Cada buffer é mapeado para uma região única da memória física, com o endereço base de cada buffer sofrendo um deslocamento de **`i * screen_size`** para garantir que não há sobreposições.
    - A utilização do triple buffering permite a eliminação de flickering, já que novos frames não são escritos diretamente no buffer do ecrã.
    - O page flipping é utilizado de forma a evitar a cópia por inteiro do buffer com o próximo frame, informando apenas ao GPU a localização deste buffer na memória RAM.

- **Integração de notificações (*banners*) entre dois jogadores, tendo como veículo a serial port**

  Através da utilização da serial port, implementamos um "acompanhamento" da partida de outros jogadores. Ao correr o jogo simultaneamente em dois sistemas, recebe-se uma notificação, em forma de "banner", a relatar acontecimentos nas rondas de outros jogadores, com mensagens como: "A player has won €10", "A player has lost €200" ou "A player has tied". As notificações aparecem em qualquer estado, e têm duração de 3 segundos.

- **Caixa de texto para a inserção do valor da aposta**

No estado **GAME_BET**, o jogador pode selecionar o *input* com o rato e escrever uma aposta numérica utilizando o teclado. A introdução das teclas numéricas indexa um array do tipo **xpm_map_t**, permitindo o desenho do número em questão. Também implementamos o "corrigir" com a tecla **BACKSPACE**.

Menu de inserção de aposta

2.2. Funcionalidades não implementadas

- **Multiplayer**

Devido à complexidade do jogo e à forma como o jogo foi implementado, converter o jogo para ser jogável em multijogador não era viável. Assim, optámos por adicionar um sistema de notificações dinâmicas entre computadores já descrito em um tópico acima.

Como um jogo de Blackjack nunca teria contacto direto entre jogadores, acreditamos que com a implementação atual atingimos a mesma funcionalidade, provando a utilidade da serial port, sem causar confusão aos jogadores.

2.3. Dispositivos Usados e o Seu Propósito

Dispositivo	**Propósito**	**Interrupções**
Timer	Distribuição de cartas, atualização da gráfica, atualização do frame das animações, e controlo dos banners. Gestão de interrupções em geral.	Sim
Teclado	Introdução de *input*; A tecla **ENTER** para iniciar o jogo, a tecla ESC para sair da aplicação e as teclas numéricas para seleção de botões.	Sim
Rato	Clicar em botões de opção e selecionar a secção do input pré-submissão.	Sim
Gráfica	Interface gráfica para a aplicação	Não
Serial Port (UART)	A serial port permite receber notificações sobre o fim de rondas noutro computador.	Sim
Real-Time Clock (RTC)	O RTC dá 50 fichas a cada 5 minutos. (Baseado na hora atual, por exemplo: 18:05, 18:10, 18:15, etc..)	Sim

A implementação dos dispositivos, à exceção da Serial Port e do Real-Time Clock, baseou-se fortemente nos labs das aulas práticas. Evidentemente, a sua integração, dadas as necessidades do jogo, só foi conseguida através das adaptações referidas abaixo.

2.3.1 Timer

Implementado por: **Todos**
Peso Relativo: **4%**

A implementação do timer é idêntica à do Lab2.

No jogo, o timer é importante para:

- Automatização de animações da lógica, nomeadamente, na alternância entre a vez do **jogador,** definida em ev_game_play.c (estado **GAME_PLAY** que aguarda, em busy waiting, a escolha do jogador, submetida como input através dos outros dispositivos), e a vez do **dealer**, uma sequência automática de transições que consiste em hits sucessivos (retiradas de cartas do baralho), definida no ficheiro **ev_game_dealer.c** Para isto, ambos os módulos usam a variável externa timer_counter, definida no ficheiro **timer.c**.
- Controlo do tempo de *display* da notificações/banners, tanto as de aviso, que acontecem, por exemplo, quando o jogador insere uma aposta com um valor que ultrapassa o seu saldo, bem como as causadas pela serial port. Estas notificações permanecem no ecrã através do incremento da variável **banner->current_timeout**, em cada interrupção do timer. Uma vez que ela seja igual à variável **banner->timeout**, a notificação desaparece.

Estes mecanismos não só estão implementados nos módulos dos drivers respectivos, mas também na lógica de jogo, bem como nas classes **app** e **game**, e são chamados na função **handle_general** do módulo **event_listener**, que também usa a variável externa **timer_counter**.

2.3.2 Teclado

Implementado por: **Todos**
Peso Relativo: **5%**

A implementação do teclado é idêntica à do Lab3.

No jogo, o teclado é importante para a receção do input, o qual é utilizado para navegar dentro da aplicação e para o preenchimento de caixas de texto como, por exemplo, o formulário do valor da aposta. A variável externa **scancode**, como o timer_counter no exemplo anterior, é passada para os diferentes ficheiros do módulo de eventos (**ev_listener**), sendo testada em condições de igualdade.

2.3.3 Rato

Implementado por: **João Santos**
Peso Relativo: **7%**

A implementação do rato, embora idêntica à do Lab4, foi fonte de alguns problemas, pois tínhamos um erro de sync dos pacotes recebidos, causando outputs inesperados. Entretanto, com algum esforço, acabamos por conseguir resolver o problema.

No jogo, tal como o teclado, o rato é importante para a receção de input, nomeadamente na interação com botões, que faz o utilizador conseguir navegar entre páginas e executar certas ações, através da verificação de **colisão** entre o **cursor** e as diferentes **sprites**. As diferentes funcionalidades do rato, dependendo do estado atual da aplicação, são todas feitas também no **ev_listener**.

2.3.4 Gráfica

Implementado por: **Todos**
Peso Relativo: **7%**

Esta implementação é semelhante à do Lab5, embora assente sobre uma estrutura que contém toda a informação sobre o modo gráfico, **gph_t**.

Para este projeto, usamos sempre o mesmo modo gráfico, **0x14C** (Cores diretas com 32 bits, resolução 1152x864).

Por questoẽs de praticidade, em vez de termos apenas a função **vg_draw_line**, que imprimia uma linha horizontal, agora temos também para linhas horizontais **vg_draw_hline** e verticais **vg_draw_vline**.

A função **vg_draw_rectangle** é usada, por exemplo, no display do título e das notificações. Também a função **vg_draw_border** é usada nos mesmos casos.

A estrutura **gph_t** contém ainda uma variável booleana, **needs_redraw**, e o seu setter, **vg_set_redraw**, e getter, **vg_has_redraw**. Esta variável é posta a 1 quando a gráfica precisa ser redesenhada. Isto permite a comunicação com outras funções que usam a gráfica: nomeadamente, a lógica mais complexa desenvolvida no módulo **drawer.c.**

A função **vg_flip** implementa o **page flipping**, com **triple buffering**, de forma a eliminar o **flickering** e o **tearing** do ecrã.

2.3.5 UART

Implementado por: **Gabriel Lima**
Peso Relativo: **4%**

O UART ou Universal Asynchronous Receiver/Transmitter é um dispositivo que permite a comunicação entre dois aparelhos através de mensagens (neste caso com o tamanho de 1 byte/8 bits).

A implementação do UART utiliza uma mistura de interrupções e “polling” para ler/escrever dados. Adicionalmente a funcionalidade FIFO (estrutura igual à fila) é usada para evitar a perda de informação e reduzir o número de interrupções necessárias. Este módulo utiliza duas filas, uma com o conteúdo a enviar (*transmitter*) e outra com o conteúdo recebido (*receiver*).

Para enviar uma mensagem é utilizada a função **uart_send_byte**. Esta coloca o conteúdo a enviar na fila de envio (*transmitter*) e utilizando polling tenta escrever o conteúdo da fila, byte a byte, no registro THR. Isto ocorre até a fila de envio estar vazia ou o estado obtido no registro LSR (*Line Status Register*) indicar que o registro THR (*Transmitter Holding Register*) está cheio, independente de qual ocorrer primeiro (**uart_fifo_write**). Ao receber uma interrupção indicando que o registro THR encontra-se vazio, é tentado novamente a escrita do conteúdo da fila de envio neste registro. (função)

Dois tipos de interrupção indicam o recebimento de uma ou mais mensagens, *Transmitter Holding Register Empty* e *Character Timeout (FIFO)*. Ao recebê-los, é tentada a leitura em polling do registro RBR (*Receiver Buffer Register*) até o estado obtido no registro LSR indicar que não contém mais dados (**uart_fifo_read**). Se houver alguma mensagem lida corretamente, esta será adicionada à fila de recebimento (*receiver*). Os conteúdos deste fila poderão ser acessados através da função **uart_get_byte**, a qual devolve o conteúdo recebido byte a byte, pela ordem de recebimento.

A função **uart_init** configura o dispositivo para receber interrupções de mensagem recebida (registro RBR com conteúdo), registro THR vazio. Adicionalmente ativa a FIFO, limpando a FIFO de envio e de transmissão, e muda o nível de interrupções de recebimento para nível 1 (interrupção por byte recebido). As mensagens são alteradas para serem compostas por 8 bits, 1 bit de paragem (stop bit) e sem paridade.

2.3.6 RTC

Implementado por: **Sara Cortez**
Peso Relativo: **3%**

O RTC é um dispositivo que assenta na comunicação entre três registos, utilizando as interrupções de alarme (*alarm*) e atualização (*update*). O mesmo é utilizado tanto para atualizar o relógio presente no ecrã, como para oferecer ao jogador, de 5 em 5 minutos, uma compensação de €50.

3.Estrutura do Código

3.1 Módulos desenvolvidos

```
# name of the program (Minix service)
PROG=proj

# source code files to be compiled
SRCS = proj.c
# drivers
SRCS += dvr_graphics.c dvr_kbc.c dvr_keyboard.c dvr_mouse.c dvr_timer.c dvr_rtc.c dvr_uart.c utils.c
# data_structs
SRCS += queue.c stack.c
# models
SRCS += sprite.c app.c player.c card.c game.c cursor.c animation.c font.c banner.c com_manager.c
#listeners
SRCS += ev_listener.c ev_game_bet.c ev_game_dealer_turn.c ev_game_over.c ev_game_play.c
ev_main_menu.c
#others
SRCS += drawer.c
```

Makefile

A pasta **/src** do diretório principal divide-se nas seguintes pastas compiláveis:
/data_structures – estruturas de dados implementadas para auxílio da lógica:

- **/stack** - estrutura inicialmente utilizada para imprimir o conteúdo da bet, mas substituída pela impressão de dígitos através do módulo **font**.
  ficheiros: stack.c, stack.h
- **/queue** - utilizado na implementação do baralho de cartas, nos menus de botões, e na implementação da serial port.
  módulos: queue.c, queue.h

/drawer - O módulo responsável por estabelecer o diálogo entre a lógica de jogo e a interface gráfica.
ficheiros: drawer.c, drawer.h
/drivers - O módulo onde estão todos os dispositivos

- **/graphics** - ficheiros: dvr_graphics.c, graphics.h, iVBE.h
- **/input** - ficheiros: dvr_kbc.c, dvr_keyboard.c, dvr_mouse.c, i8042.h, input.h
- **/rtc -** ficheiros: dvr_rtc.c, rtc.h
- **/serial_port -** ficheiros: dvr_uart.c, iUART.h, serial_port.h
- **/utils** - ficheiros: utils.h, utils.c
- **/timer -** ficheiros: dvr_timer.c, i8254.h, timer.h
  drivers.h

/ev_listener - Divisão do jogo em estados, e handling de interrupções de acordo com o estado atual. As suas funções integram o estado no loop **driver_receive** do módulo principal. Os ficheiros envolvidos neste módulo são: ev_game_bet.c, ev_game_dealer_turn.c, ev_game_over.c, ev_game_play.c, ev_listener.c, ev_listener.h, ev_main_menu.c

/model - O módulo onde estão todos os componentes da aplicação e a definição da maior parte das structs lógicas usadas. Cada um dos subdiretórios contém apenas os módulos homónimos .c e .h.

/animation
/app
/banner
/card
/com_manager
/cursor
/font
/game
/player
/sprite

proj.c - Inicialização, loop driver_receive, e shut down da app

O módulo **/assets** contém xpms para **button**, **card**, **font** e o **background**, não se destinando à compilação.

3.1.1 Assets

Implementado Por: **João Santos** e **Sara Cortez**
Peso Relativo: **3%**

Todos os pixmaps usados.

3.1.2 Data Structures

Implementado Por: **Gabriel Lima** e **João Santos**
Peso Relativo: **5%**

- **Queue**

Usámos a fila na implementação do baralho de cartas, nos menus de botões, e na implementação da serial port.

A fila é uma C-struct que assenta um array de memória dinâmica. A função **queue_create** é responsável pela inicialização da fila, alocando memória para a estrutura da mesma e para o array interno que armazenará os elementos. Em caso de falha na alocação de memória, a função garante a limpeza adequada dos recursos alocados.

A função **queue_size** retorna o número atual de elementos na fila, enquanto **queue_full** e **queue_empty** verificam se a fila está cheia ou vazia, respectivamente, assegurando a integridade das operações subsequentes.

Já a função **queue_destroy** liberta a memória ocupada pelos elementos da fila, utilizando uma função fornecida para libertar a memória de cada elemento individualmente, antes de libertar a própria estrutura da fila.

Para aceder a um elemento específico da fila, **queue_at** calcula a posição correta no array, considerando o comportamento circular da fila. A função **queue_at_ref** retorna uma referência ao elemento, facilitando a manipulação direta dos dados.

A remoção do elemento da frente da fila é feita por **queue_pop**, que ajusta os índices de forma a manter a continuidade circular da fila. Já a inserção de novos elementos é gerida por **queue_push**, que coloca o novo elemento na posição correta, ajustando os índices e verificando se a fila está cheia ou se o conteúdo é inválido.

Finalmente, a função **queue_shuffle** "baralha" os elementos na fila de forma aleatória, garantindo a aleatoriedade das posições dos elementos. Esta função é útil para a implementação do baralho de cartas.

- **Stack**

Inicialmente planeámos usar a stack para o input, para apagar e inserir com facilidade. Depois percebemos que, dada a criação do módulo da **font** (o qual consegue "imprimir" números no ecrã), tal estrutura já não era necessária. Decidimos mantê-la para potencial necessidade, mas acabamos por não a usar.

A implementação da pilha, tal como a da fila, compõe-se de uma C-struct assente num array de memória dinâmica, com parâmetros para o tamanho atual, o tamanho base e um apontador para a posição final.

Funções como **stack_create**, **stack_size**, **stack_full**, **stack_empty, stack_destroy**, **stack_pop**, **stack_push** e **stack_at** têm propósitos análogos.

3.1.3 Model

Os submódulos do **model** definem os diferentes componentes do jogo, sendo a comunicação entre elas articulada com o módulo **Event Listener**.

3.1.3.1 Sprite

Implementado Por: **João Santos** e **Gabriel Lima**
Peso Relativo: **6%**

A estrutura **sprite_t** encapsula as propriedades de uma sprite, incluindo suas coordenadas (x, y), imagem (xpm_image_t) e mapa de píxeis (map). A função **sprite_create** cria uma nova sprite, carregando a imagem a partir de um mapa XPM, inicializando as coordenadas para zero.

A função **sprite_queue_destroy** serve como wrapper para destruir um sprite quando usado em conjunto com filas, chamando **sprite_destroy**, que liberta a memória associada ao mapa de pixels do sprite.

Para desenhar a sprite no ecrã, a função **sprite_draw** itera sobre o mapa de píxeis da imagem e desenha cada pixel não transparente na posição correta no ecrã. Utiliza **vg_draw_pixel** para desenhar cada píxel e **vg_get_info** para obter informações gráficas como a profundidade de cor e resolução. A função **sprite_draw_force_color** é similar a sprite_draw, mas força todos os píxeis não transparentes a serem desenhados com uma cor específica fornecida como argumento. Para desenhar o sprite com rotação de 90º, **sprite_draw_rotate** também itera sobre o mapa de pixels, mas ajusta as coordenadas de desenho para realizar a rotação, desenhando cada pixel na posição adequada.

O movimento da sprite é gerido pela função **sprite_move**, que atualiza as coordenadas (x, y) do sprite. A função verifica se as novas coordenadas estão dentro dos limites do ecrã para evitar erros.

A função **sprite_colides** verifica se dois sprites colidem, comparando as coordenadas de seus limites. Se as áreas dos dois sprites se sobrepuserem, a função retorna um valor indicador de colisão.

3.1.3.2 Animation

Implementado Por: **Marta Silva** e **Gabriel Lima**
Peso Relativo: **4%**

A estrutura **animation_t** contém quatro filas: uma para frames de sprites, duas para coordenadas **x** e **y**, e a última para flags de rotação. Contém ainda o índice do frame atual da animação e uma função de callback, **void (*on_end)(void *)** a ser chamada quando a animação termina.

A função **animation_create** funciona como um construtor, recebendo como argumento o tamanho das filas, **frame_amount**, e a função de callback. A função **animation_add_frame** adiciona uma frame à animação, inserindo em todas as filas. Já **animation_destroy** atua como um destrutor.

Para desenhar a frame atual, **animation_draw** obtém as coordenadas, a sprite e a flag de rotação das filas, através de chamadas a funções de **sprite**. A execução da animação é gerida pela função **animation_run**, que incrementa o índice do quadro atual e verifica se a animação chegou ao fim. Se todos os frames tiverem sido exibidos, a função chama **on_end** e destrói a animação.

3.1.3.3 Player

Implementado Por: **Gabriel Lima** e **João Santos**
Peso Relativo: **6%**

O módulo player mimetiza um jogador, gerindo as operações e estado relacionados com ele relacionados. A estrutura **player t** é utilizada para representar um jogador e contém informações sobre o mesmo, como as suas moedas, apostas, valor das cartas, moedas ganhas, o estado de **game over** do jogo e a fila de cartas.

Para inicializar um jogador, a função **player_init** define os valores iniciais das moedas, apostas e valor das cartas, e cria uma fila de cartas com um tamanho máximo definido por **PLAYER_MAX_DECK_SIZE**. A função retorna um valor de erro se a alocação da fila falhar.

A destruição de um jogador está a cargo da função **player_destroy**, que chama **queue_destroy** para destruir a fila de cartas e a função específica **card_queue_destroy**. Isto garante que todos os recursos alocados para as cartas do jogador sejam libertados corretamente.

A função **player_draw** é responsável por desenhar as cartas do jogador no ecrã, iterando sobre a fila de cartas, calculando a posição de cada carta baseada no índice e no tamanho da imagem da carta. Se a carta estiver marcada como **is_double**, é desenhada com rotação usando **sprite_draw_rotate**. Caso contrário, é desenhada normalmente com **sprite_draw**. A existência desta variável tem a ver com a animação rotativa que acontece quando o utilizador toma a ação **double**.

A enumeração **game_over_state_t** define os possíveis estados do jogo (não terminado - default, vitória do jogador, derrota do jogador e empate), permitindo uma verificação fácil e manipulação do estado do jogo para o jogador.

3.1.3.4 Game

Implementado Por: **Gabriel Lima** e **João Santos**
Peso Relativo: **6%**

Este módulo implementa a gestão do jogo, incluindo a criação do baralho, inicialização do jogo, distribuição de cartas, desenho das cartas no ecrã e cálculo do valor total das cartas.

A estrutura **game_t** representa o estado atual do jogo e contém várias informações importantes, como se pode ver abaixo:

```
typedef struct {
  sprite_t *card_back;              /**< Sprite for the back of the cards. */
  queue_t *cards;                   /**< Queue of cards for the game deck. */
  queue_t *dealer;                  /**< Queue of cards for the dealer. */
  animation_t *curr_anim;           /**< Pointer to the current animation. */
  uint32_t dealer_value;            /**< Value of the dealer's cards. */
  uint8_t input_select;             /**< Input select flag. */
  uint8_t dealer_turn;              /**< Flag indicating if it's the dealer's turn. */
  uint8_t dealer_ignore_last;       /**< Flag to ignore the dealer's last card. */
  player_t main_player;             /**< Main player of the game. */
  player_t other_player;            /**< Other player in the game. */
} game_t;
```

A função **game_add_deck** adiciona um baralho ao jogo, criando cartas para cada combinação de valor e naipe e inserindo-as na fila de cartas especificada. O parâmetro **shuffle** determina se o baralho deve ser baralhado após ser criado, e se for, chama **queue_shuffle**.

A função **game_init** inicia o jogo, definindo os valores iniciais para as variáveis do jogo, criando os sprites das cartas, as filas de cartas para o baralho e o dealer, e iniciando o jogador. A função **game_destroy** liberta a memória alocada para todos os recursos do jogo.

As funções **game_give_card** e **game_draw_deck** são responsáveis por distribuir cartas do baralho para os jogadores e para a mesa, respectivamente. A função **game_draw_dealer** desenha as cartas do dealer no ecrã, exibindo apenas a primeira carta quando não é a vez do dealer jogar.

Por fim, a função **game_get_cards_value** calcula o valor total das cartas de uma fila de cartas, considerando as regras do jogo (como o valor dos ases).

3.1.3.5 COM Manager (Serial Port)

Implementado Por: **Gabriel Lima**
Peso Relativo: **3%**

O COM manager gerencia o recebimento de mensagens da serial port. Quando a serial port recebe um byte, a função **com_add_byte** é chamada e tenta adicionar o byte a uma estrutura (**com_packet_t**) que representa uma mensagem. Esta estrutura é composta por um tipo (**proto_msg_type_t**) e pelo conteúdo da mensagem, um número de 32 bits (**uint32_t**). Ao receber informação suficiente para preencher a estrutura, a função **com_handle_packet** é executada e aplica a informação ao estado atual da aplicação.

Como a serial port só consegue enviar 1 byte de cada vez, este módulo providência a função **com_send_msg**, a qual divide a mensagem em 5 bytes (+ 2 do header byte e do trail byte do nosso protocolo de comunicação).

3.1.3.6 Cursor

Implementado Por: **João Santos** e **Gabriel Lima**
Peso Relativo: **4%**

A estrutura **`cursor_t`** representa o cursor e inclui informações sobre suas coordenadas (x e y), sprites para o cursor em diferentes estados **cursor_state_t** (**`POINTER`** ou **`HAND`**), e o estado atual do cursor (**`state`**).

A função **`cursor_init`** inicializa o cursor, definindo o estado como **POINTER** (ponteiro), configurando as coordenadas iniciais para o centro do ecrã e criando os sprites para o cursor em cada estado. Se a criação de qualquer sprite falhar, a função retorna um código de erro.

A função **`cursor_move`** move o cursor para uma nova posição especificada pelas coordenadas **`x`** e **`y`**. Se as coordenadas estiverem fora dos limites do ecrã, a função retorna um código de erro.

A função **`cursor_draw`** desenha o cursor no ecrã de acordo com o estado atual. Se o estado for **POINTER**, o sprite do ponteiro é desenhado na posição atual do cursor; se for **HAND**, o sprite da mão é desenhado.

As funções **`cursor_sprite_colides`** e **`cursor_box_colides`** verificam se o cursor colide com um sprite específico ou com uma caixa delimitada por coordenadas (**x**, **y**, **x2**, **y2**). Se o cursor estiver no estado **POINTER** e houver uma colisão com o sprite ou a caixa delimitada, o estado do cursor é alterado para **HAND**. A função retorna verdadeiro se houver uma colisão e o botão esquerdo do mouse estiver pressionado; caso contrário, retorna falso.

3.1.3.7 Font

Implementado Por: **Gabriel Lima**, **João Santos** e **Sara Cortez**
Peso Relativo: **4%**

Font serve para imprimir caracteres, strings e números no ecrã usando sprites correspondentes a cada caractere. A estrutura **font_t** contém duas **queue**s de sprites, uma para os números e outra para as letras, além de sprites para o símbolo do euro e de dois pontos.

A função **font_init** inicia a fonte, criando sprites para cada caractere alfanumérico e adicionando-os às filas correspondentes. Também cria sprites para o símbolo do euro e dois pontos.

A função **font_print_str** imprime uma string no ecrã, percorrendo cada caractere e chamando **font_print_char** para desenhar cada caractere individualmente. Para além disso, a também ajusta a posição de impressão com base no caractere seguinte, para garantir um espaçamento adequado entre os caracteres. A função **font_print_number** converte um número numa string e chama **font_print_str** para imprimir o número no ecrã.

```
void font_print_char(font_t *font, char c, uint32_t x, uint32_t y, uint32_t color)
{
  sprite_t *sprite = NULL;
  if (c >= 'a' && c <= 'z')
  {
    sprite = queue_at(font->xpms_letters, c - 'a');
  }
```

3.1.3.8 Card

Implementado Por: **Gabriel Lima** e **João Santos**
Peso Relativo: **5%**

Este código implementa um módulo para manipular cartas em um jogo de cartas. Ele define uma estrutura, **`card_t`**, que representa uma carta, com um valor, naipe, indicador **is_double** para se saber se se usa a animação própria de jogadas

DOUBLE, e um ponteiro para o sprite da carta, que se encontra na matriz (naipes x valores) **card_sprites**.

A função **`card_base_init`** inicia o conjunto de sprites de cartas, carregando os sprites para cada valor e tipo de carta.

A função **`card_create`** aloca memória para a carta, define seu valor e tipo, e associa a sprite correspondente da matriz **`card_sprites`** à carta.

Há três destrutores para usar nos diferentes contextos: **`card_destroy`** liberta a memória alocada para uma simples carta, enquanto **`card_base_destroy`** liberta a memória alocada de todas as sprites da mesma. A função **`card_queue_destroy`** é usada na função **queue_destroy** para eliminar uma fila de cartas.

3.1.3.9 Banner

Implementado Por: **Gabriel Lima**
Peso Relativo: **3%**

A estrutura **banner_t** é utilizada para gerir um banner de notificações. Cada banner contém uma mensagem, um tempo limite para exibição e um contador para controlar o tempo decorrido desde o início da exibição da mensagem.

A função **banner_init** é responsável por inicializar a estrutura do banner, garantindo que todos os seus campos estejam devidamente configurados.

A função **banner_set_message** permite alterar um banner que já exista, definindo uma nova mensagem e tempo limite para exibição. Esta função também solicita uma atualização do ecrã para garantir que o novo banner seja exibido imediatamente.

A função **banner_update_timeout** é chamada periodicamente para atualizar o contador de tempo decorrido desde o início da exibição da mensagem. Quando o tempo limite é atingido, a função solicita uma nova atualização do ecrã e remove a mensagem do banner.

Já a função **banner_draw** é responsável por desenhar o banner no ecrã, utilizando as dimensões e a mensagem definidas na estrutura do banner. Isso é feito através de chamadas a funções de desenho de retângulos de texto, garantindo que o banner seja exibido de forma centrada e legível.

3.1.3.10 App

Implementado Por: **Todos**
Peso Relativo: **6%**

A estrutura **`app_t`** é o cerne desta aplicação. Ela encapsula todos os elementos essenciais para gerir o estado da aplicação, desde o cursor até os componentes da interface do utilizador e o estado do jogo. É aqui que se coordenam todas as operações e transições de estado.

A enumeração **app_state_t** define os diferentes estados que a aplicação pode assumir, desde o menu principal até o fim do jogo.

Dentro da estrutura **app_t**, podemos encontrar uma variedade de elementos, cada um desempenhando um papel fundamental no funcionamento da aplicação. Por exemplo, o objeto **cursor** controla a posição do cursor no ecrã, enquanto os sprites do **background** e dos botões são responsáveis por criar a interface visual que os utilizadores interagem.

```
 */
typedef struct {
  cursor_t cursor;                  /**< Cursor object. */
  sprite_t *background;             /**< Background sprite. */
  queue_t *buttons_main_menu;       /**< Queue of main menu buttons. */
  queue_t *buttons_game_playing;    /**< Queue of game playing buttons. */
  queue_t *buttons_game_over;       /**< Queue of game over buttons. */
  sprite_t *button_bet;             /**< Bet button sprite. */
  font_t font;                      /**< Font object. */
  banner_t banner;                  /**< Banner object. */
  game_t game;                      /**< Game state object. */
  app_state_t state;                /**< Current application state. */
} app_t;
```

3.1.4 Drawer

Implementado Por: **Todos**
Peso Relativo: **5%**

A função **draw_state** é responsável por desenhar o estado atual da aplicação no ecrã. Com base no estado da aplicação, ela desenha os elementos correspondentes, como o banner, os botões e informações do jogo.

Por exemplo, quando a aplicação não está no estado de menu principal, o estado do jogo é desenhado no ecrã. Isso inclui o saldo atual do jogador, as suas cartas e as ações disponíveis, como **HIT**, **STAND**, **DOUBLE** e **FORFEIT**. Além disso, o valor da aposta e a pontuação das cartas também são exibidos de forma clara e organizada.

De seguida, a função verifica o estado específico da aplicação e desenha os elementos correspondentes. Por exemplo, quando o estado do jogo é **GAME_BET**, a mensagem "Please place your bet" é exibida no ecrã juntamente com o botão de aposta. O destaque é dado ao botão de aposta quando selecionado pelo jogador, garantindo uma experiência visual intuitiva.

Finalmente, a função **draw_button_set** é usada para posicionar e desenhar os conjuntos de botões no ecrã. Ela calcula a posição de cada botão com base na largura do ecrã e no número de botões no conjunto, garantindo que eles sejam espaçados uniformemente e exibidos de forma adequada.

3.1.5 Devices

Resumido na secção anterior.

3.1.6 Event Listener

Implementado Por: **João Santos**, **Gabriel Lima**, **Marta Silva**
Peso Relativo: **7%**

O diretório **ev_listener** contém funções responsáveis por lidar com as interrupções dos dispositivos e traduzi-las na lógica de jogo. A função **handle_interrupt** é o ponto central para a gestão das interrupções, chamando a função específica para o estado atual da aplicação, representada pelos handlers contidos no array listeners. Ela chama a função **handle_general**, que consiste num switch-case para lidar com interrupções gerais, como teclado, mouse, UART, temporizador e RTC, e depois chama a função específica para o estado atual da aplicação, representada pelos handlers contidos no array listeners.

Como existem muitas funções para gerir cada um dos estados, optámos por dividi-las em vários ficheiros: **ev_game_bet.c**, **ev_game_dealer_turn.c**, **ev_game_over.c**, **ev_game_play.c**, **ev_main_menu.c** e **ev_listener.c**.

3.2 Proj

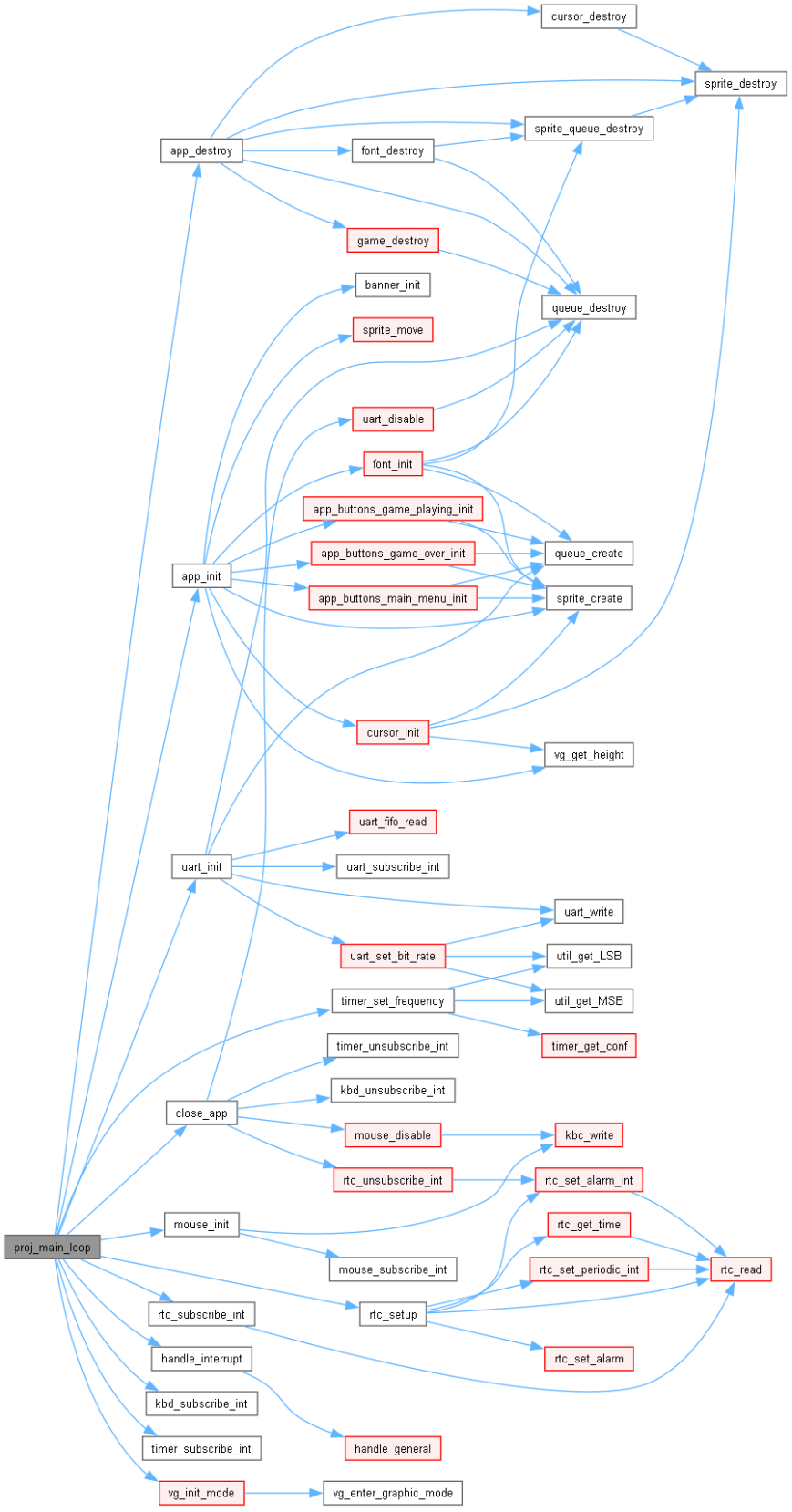
Implementado Por: **João Santos**
Peso Relativo: **3%**

Este módulo contém o loop principal da aplicação. Começa com a subscrição de todas as interrupções necessárias para as drivers, corre o loop e, na função closeApp(), cancela a subscrição de interrupções, terminando a aplicação.

```
while (app->state != EXIT)
{
  if (driver_receive(ANY, &msg, &ipc_status)) continue;

  if (!is_ipc_notify(ipc_status)) continue;

  if (_ENDPOINT_P(msg.m_source) != HARDWARE) continue;

  if (msg.m_notify.interrupts & bit_no.mouse)
  {
    handle_interrupt(app, MOUSE);
  }

  if (msg.m_notify.interrupts & bit_no.uart)
  {
    handle_interrupt(app, UART);
  }

  if (msg.m_notify.interrupts & bit_no.rtc)
  {
    handle_interrupt(app, RTC);
  }

  if (msg.m_notify.interrupts & bit_no.kb)
  {
    handle_interrupt(app, KEYBOARD);
  }

  if (msg.m_notify.interrupts & bit_no.timer)
  {
    handle_interrupt(app, TIMER);
  }
}
```

Loop Driver Receive.

3.3 Gráfico de chamada de funções



4. Detalhes de Implementação

4.1 Fluxo de Estados

As quatro fases referidas acima foram implementadas como quatro estados na lógica de jogo:
1. **MAIN_MENU** - O estado inicial, a apresentação do jogo: contém opções de início (**START**) de uma ronda e de saída (**EXIT**) da aplicação.

2. **GAME_BET** - Este estado aparece mal o jogo começa. Contém o campo para input da aposta, que é um número entre 1 e o saldo atual do jogador, que podemos encontrar no canto inferior esquerdo. A lógica não permite que o input ou o doubling da amostra ultrapasse esse valor. Para Bet podemos também clicar **ENTER**.

3. **GAME_PLAY** - Após a inserção do valor da aposta, as cartas são "dadas" com uma animação que é sempre igual. O valor que o jogador apostou e o saldo, já sem esse valor, estão no canto inferior esquerdo. O estado onde o jogador escolhe uma das quatro opções **HIT**, **STAND**, **DOUBLE** ou **FORFEIT**. Premir, respetivamente , os botões 1, 2, 3, ou 4 no teclado, é o equivalente a clicar nos botões com o rato. A animação consequente depende da opção escolhida.
    5. **HIT** - Dá mais uma carta ao jogador. Só transita para um estado de **GAME_DEALER_TURN**, se as cartas do player excederem ou igualarem 21.

6. **STAND** - transita para um estado de **GAME_DEALER_TURN** e o player fica com valor atual das cartas.
7. **DOUBLE** - O número da aposta duplica, e a carta extra cobre as que o player tinha.
8. **FORFEIT** - A única que transita de estado sem uma animação nas cartas.

4. **GAME_DEALER_TURN –** Sucessivas animações para hits do dealer.

4. **GAME_OVER** - Podemos ver este estado como "abstrato", visto que se concretiza num dos seguintes subestados:

9. **PLAYER_WIN** (vitória) - O valor da aposta é devolvido, em dobro, ao jogador.
10. **PLAYER_LOSE** (derrota) - O jogador perde o valor da aposta.
11. **PLAYER_DRAW** (empate) - O jogador recebe de volta o valor da aposta.
12. **PLAYER_SURRENDER** (desistência) - O jogador recebe de volta metade do valor da aposta.

Neste estado são mostrados os botões de **REBET** ou de **EXIT**. Clicar com o rato em **REBET** ou premir **ENTER** redireciona para o **GAME_BET**, mas agora o saldo com que o jogador começa é aquele com que o jogador terminou na ronda anterior.

5. Conclusão

Para concluir, estamos bastante satisfeitos com o projeto e acreditamos que conseguimos superar as expectativas que nos foram propostas. No entanto, claro que há sempre certas coisas que gostaríamos de melhorar. Alguns exemplos destas seriam fazer animações mais complexas e quem sabe arranjar algum tipo de sistema de depósito e levantamento de moedas, de modo a dar um pouco mais de emoção ao jogo.

Estamos bastante orgulhosos pelo que conseguimos atingir no trabalho, especialmente pelo rigor e perfeição que conseguimos manter ao longo do projeto todo, tendo assim atingido um código legível e organizado com um design agradável e com todas as funcionalidades sem erros e bugs conhecidos.

Acima de tudo, conseguimos completar o nosso objetivo principal, que era proporcionar um jogo que proporcionasse vontade ao jogador de continuar a jogar.

Algo que também se refletiu bastante ao longo do trabalho foi o quão mais motivante é trabalhar com pessoas que estão interessadas e puxam-te a ir mais longe.