Final Project Design Document

Yang Li yl6070, Jiyang Yin jy3557, Zhewen Guo zg2567

April 28, 2026

1 Introduction and Overview

1.1 Project Summary

This project implements a two-player fighting game prototype on the `DE1-SoC` platform. The current system combines HPS-side game logic with FPGA-side video and audio output:

- The `HPS` side in C handles USB keyboard and Linux gamepad input, game-state updates, collision and damage logic, animation selection, software scene composition, audio command scheduling, and MMIO access.
- In addition to USB keyboard input, the final system includes a Linux `evdev` gamepad input path. The gamepad module reads `/dev/input` event devices nonblocking, converts joystick axes and button events into the same `fighter_player_result_t` structure used by the keyboard parser, and therefore shares the existing gameplay logic.
- The primary video path is now an FPGA VGA framebuffer peripheral, `fighter_vga_renderer`. HPS software renders a `320x240` RGB565 framebuffer and writes it through Avalon-MM. The FPGA scales it to `640x480` VGA timing and performs vblank-safe double-buffer swaps.
- The video code still includes fallbacks to Linux framebuffer devices such as `/dev/fb0` and to console output when the FPGA VGA path is unavailable.
- The FPGA side implements a `WM8731` audio peripheral, `fighter_audio_wm8731`, with FIFO-backed sample writes, I2C codec initialization, and serial DAC output.
- The software still contains `fighter_mmio_encode()`, which packs game state into 22 words. In the current hardware design, however, the implemented VGA IP is framebuffer-based rather than a sprite engine that directly consumes those 22 game-state words.

1.2 Demo Video Link and Screenshot

https://drive.google.com/file/d/1ZNdd4ZHMPT2DrSgyldJLVKlHCBUpnBR4/view?usp=sharing

Figure 1: Screenshots of the final Street Fighter FPGA system running on the VGA display.

1.3 Relationship Between Current Implementation and Target Architecture

Layer	Current Status	Notes
Dual USB keyboard input	Implemented	`libusb` polling for up to two HID boot keyboards when `libusb-1.0` is available
Fighting game state machine	Implemented	Menu, match, game over, KO, timeout, exit, blocking, projectiles, hit and block resolution
Animation and assets	Implemented	Ryu and Ken PPM sprite sets, background image, menu images, and fireball projectiles
FPGA VGA framebuffer renderer	Implemented	`fighter_vga_renderer` at default physical address `0xFF240000`
Linux framebuffer / console video fallback	Implemented	Used if VGA MMIO probing fails or if console mode is requested
FPGA audio peripheral	Implemented	`fighter_audio_wm8731` at default physical address `0xFF200000`
Game-state MMIO encoder	Implemented in software	Exports 22 32-bit words, but current VGA hardware uses a framebuffer protocol instead

1.4 Assumptions and Constraints

- The target board is `DE1-SoC`.
- The system assumes up to two USB keyboards by default, with an optional Linux `evdev` gamepad path for joystick-style controllers.
- The default keyboard input mapping is `W/A/S/D/J/K/L`.
- The main logic runs at approximately 60 FPS.
- The game coordinate system is `640x480`. The FPGA framebuffer is `320x240` and is doubled to `640x480` by hardware.
- The current design uses direct user-space physical memory mapping instead of a Linux kernel driver.

2 Top-Level System Block Diagram

2.1 Actual Current Data Path

```
USB Keyboard(s)
      |
      v
usb_hid_keyboard.c
```

```
     |
     v
fighter_input.c
     |
     +----------------------+
                            |
Gamepad / Joystick          |
/dev/input/event*           |
     |                      |
     v                      |
fighter_gamepad.c           |
     |                      |
     v                      |
fighter_player_result_t <---+
     |
     v
fighter_game.c
(game state, collision, HP, attacks, timer)
     |
     +--------------------+
     |                    |
     v                    v
fighter_animation.c       audio commands
(sprite/frame selection)  |
     |                    v
     v              fighter_audio.c
fighter_renderer.c        |
(software RGB565 frame)   |
     |                    |
     v                    v
/dev/mem MMIO       /dev/mem MMIO
LW HPS-FPGA bridge  LW HPS-FPGA bridge
     |                    |
     v                    v
fighter_vga_renderer      fighter_audio_wm8731
FPGA VGA framebuffer      FPGA audio FIFO + codec driver
     |                    |
     v                    v
VGA monitor          WM8731 audio output
```

2.2 Platform Address Map

`hw/soc_system.qsys` connects both project-specific peripherals to the HPS lightweight bridge. With the standard lightweight bridge physical base of `0xFF200000`, the current software defaults are:

IP Instance	Qsys Base	Linux Physical Address	Override Variable
`fighter_audio_0`	`0x0000`	`0xFF200000`	`FIGHTER_AUDIO_MMIO_ADDR`

IP Instance	Qsys Base	Linux Physical Address	Override Variable
fighter_vga_0	0x40000	0xFF240000	FIGHTER_VGA_MMIO_ADDR

Both VGA and audio software also access the HPS bridge reset register at 0xFFD0501C. The software clears bits [1:0] to enable the HPS-FPGA bridge before probing the custom IP.

2.3 Image and Audio Assets

Game images and sounds are stored as normal files in the HPS Linux filesystem. They are not compiled into the FPGA bitstream and they are not initialized into FPGA BRAM. The FPGA only receives final pixel samples through the VGA MMIO framebuffer and final PCM samples through the audio FIFO registers.

Asset Type	Repository Path	Runtime Loader
Ryu sprites	game_assets/sprites/RyuPPM/*/*.ppm	fighter_animation.c
Ken sprites	game_assets/sprites/KenPPM/*/*.ppm	fighter_animation.c
Fireball sprites	game_assets/sprites/*PPM/fireball/*.ppm or attack_fireball/*.ppm	fighter_animation.c
Background	game_assets/background/background.ppm	fighter_renderer.c
Menu frames	game_assets/ui/menu/menu_frame_0.ppm and menu_frame_1.ppm	fighter_renderer.c
Menu BGM	game_assets/soundeffects/Title.wav	fighter_audio.c
Confirm sound	game_assets/soundeffects/Credit.wav	fighter_audio.c
Game-over sound	game_assets/soundeffects/GameOver.wav	fighter_audio.c

The full loader source paths are sw/fighter_animation.c, sw/render_if/fighter_renderer.c, and sw/audio/fighter_audio.c.

The runtime asset search rules are:

- If FIGHTER_ASSET_ROOT is set, animation code uses $FIGHTER_ASSET_ROOT/sprites/RyuPPM and $FIGHTER_ASSET_ROOT/sprites/KenPPM. The renderer uses the same root for background and menu images.
- If FIGHTER_ASSET_ROOT is not set, animation first tries /root/game_assets/sprites/RyuPPM and /root/game_assets/sprites/KenPPM, then falls back to the repository relative paths ../game_assets/sprites/RyuPPM and ../game_assets/sprites/KenPPM.
- Background/menu loading similarly tries FIGHTER_ASSET_ROOT, /root/game_assets, and ../game_assets.
- The current WAV path function returns repository-relative paths under ../game_assets/sound effects. The MMIO audio backend opens them with fopen(), parses RIFF/WAVE chunks, accepts PCM16 mono or stereo, and resamples to the software target rate of 48000 Hz.

PPM image loading supports both `P6` binary and `P3` ASCII PPM. The software stores decoded pixels as 24-bit RGB in heap memory. When drawing to the FPGA VGA backend, the renderer packs the final composed frame into RGB565 words and writes those words through MMIO.

2.4 PPM to RGB565 Asset Pipeline

The repository also includes `scripts/convert_ppm_to_rgb565.py`, which converts editable `P6` PPM images into a packed `.rgb565` binary format. The output is explicitly RGB565, not generic RGB. This matters because RGB565 is the same 16-bit pixel format used by the VGA framebuffer. Preconverting image assets avoids decoding full 24-bit RGB images at runtime and lets the game store assets in the same format that the FPGA VGA path ultimately consumes.

The generated file layout is:

Byte Range	Meaning
`0-3`	Signature `"R565"`
`4-5`	Header size, little endian
`6-7`	Width, little endian
`8-9`	Height, little endian
`10-11`	Pixel format ID, little endian
`12-15`	Pixel data size, little endian
`16+`	Little-endian RGB565 pixels

3 Custom Peripheral Register Maps

3.1 Implemented VGA MMIO Contract

The current video hardware source is `hw/fighter_vga_renderer.sv`, packaged by `hw/fighter_vga_hw`. Its Avalon-MM interface uses 32-bit data and word-based addresses. Software therefore accesses `regs[n]` for Avalon word offset `n`.

Avalon Signal	Width	Direction	Meaning
`avs_chipselect`	1	HPS to FPGA	Selects this slave for the current transaction
`avs_read`	1	HPS to FPGA	Indicates a register read transaction
`avs_write`	1	HPS to FPGA	Indicates a register or framebuffer write transaction
`avs_address`	16	HPS to FPGA	32-bit word offset; `0`, `1`, `2`, etc., not byte address
`avs_writedata`	32	HPS to FPGA	Control word or two packed RGB565 pixels
`avs_readdata`	32	FPGA to HPS	Control/status, geometry, ident, or zero for unmapped addresses

Word Offset	Register	R/W	Description
0	REG_CONTROL	RW	Status and frame-swap control
1	REG_WIDTH	R	Returns framebuffer width 320
2	REG_HEIGHT	R	Returns framebuffer height 240
3	REG_STRIDE	R	Returns row stride 640 bytes
31	REG_IDENT	R	Returns 0x56504741, ASCII "VPGA"
1024..39423	framebuffer window	W	RGB565 framebuffer words, two pixels per 32-bit word

REG_CONTROL read bits:

- bit0: IP present, fixed to 1
- bit1: swap pending
- bit8: current display buffer
- bit9: current software write buffer

REG_CONTROL write bits:

- bit1: swap request. Software writes this after filling a full frame. Hardware performs the buffer swap at vblank and then clears swap_pending.

3.2 VGA Framebuffer Format

- Framebuffer width: 320 pixels
- Framebuffer height: 240 pixels
- Pixel format: RGB565
- Pixels per 32-bit word: 2
- Framebuffer words: 38400
- Framebuffer start word offset: 1024
- Total software mapping span: 39424 * 4 = 157696 bytes

```
word[15:0]  = first RGB565 pixel
word[31:16] = second RGB565 pixel
```

The FPGA output timing is standard 640x480. Internally, the renderer uses the 50 MHz board clock, toggles a pixel tick to generate approximately 25 MHz pixel timing, and maps each framebuffer source pixel to a 2x2 area on the VGA output.

3.3 VGA Software-Hardware Transaction Protocol

The VGA backend in `sw/render_if/fighter_renderer.c` uses the following protocol:

1. Open `/dev/mem`.
2. Map the HPS bridge reset register at `0xFFD0501C` and clear bits `[1:0]`.
3. Map the VGA MMIO span at `FIGHTER_VGA_MMIO_ADDR`, default `0xFF240000`.
4. Read `regs[31]` and require `0x56504741` (`"VPGA"`).
5. Read `regs[1]`, `regs[2]`, and `regs[3]` and require `320`, `240`, and `640`.
6. Before writing a new frame, poll `regs[0] bit1` until `swap_pending` is clear.
7. Compose the frame in software, pack every two RGB565 pixels into one 32-bit word, and write `38400` words starting at `regs[1024]`.
8. Write `regs[0] = 0x00000002`. The FPGA flips the display buffer on the next vblank.

3.4 VGA Pins

Port	Width	Direction	Data
`vga_r`	8	output	Red channel, expanded from RGB565 red bits
`vga_g`	8	output	Green channel, expanded from RGB565 green bits
`vga_b`	8	output	Blue channel, expanded from RGB565 blue bits
`vga_hs`	1	output	Active-low horizontal sync
`vga_vs`	1	output	Active-low vertical sync
`vga_clk`	1	output	Pixel clock derived by toggling at 50 MHz input clock
`vga_blank_n`	1	output	High during visible region
`vga_sync_n`	1	output	Fixed low for the DE1-SoC VGA DAC

3.5 Implemented Audio MMIO Contract

The current audio hardware source is `hw/fighter_audio.sv`, module `fighter_audio_wm8731`, packaged by `hw/fighter_audio_hw.tcl`. The default physical address is `0xFF200000`.

Avalon Signal	Width	Direction	Meaning
`avs_chipselect`	1	HPS to FPGA	Selects this audio slave
`avs_read`	1	HPS to FPGA	Reads `control`, `fifospace`, or zero from sample ports

Avalon Signal	Width	Direction	Meaning
`avs_write`	1	HPS to FPGA	Writes clear commands or sample words
`avs_address`	2	HPS to FPGA	32-bit word offset `0..3`
`avs_writedata`	32	HPS to FPGA	Control clear bits or `int16` sample in upper halfword
`avs_readdata`	32	FPGA to HPS	Status and FIFO-space fields

Word Offset	Name	R/W	Description
`0`	`control`	RW	Read FIFO and codec status; write clear commands
`1`	`fifospace`	R	`[31:24]` left write space, `[23:16]` right write space
`2`	`leftdata`	W	Left-channel sample word; software writes `int16 « 16`
`3`	`rightdata`	W	Right-channel sample word; software writes `int16 « 16`

The current `control` read word uses:

- `bit0`: `codec_init_done`
- `bit1`: `codec_init_error`
- `bit2`: `tx_underflow_seen`
- `bit3`: `write_overflow_seen`
- `bits[15:8]`: `right_count`
- `bits[23:16]`: `left_count`

The `control` write word uses:

- `bit2`: clear read/playback-side FIFO state
- `bit3`: clear write-side FIFO state

3.6 Audio Software-Hardware Transaction Protocol

The MMIO audio backend in `sw/audio/fighter_audio.c` operates as:

1. Open `/dev/mem`.
2. Map the HPS bridge reset register at `0xFFD0501C` and clear bits `[1:0]`.

3. Map four 32-bit words at `FIGHTER_AUDIO_MMIO_ADDR`, default `0xFF200000`.

4. Write `control bit2 | bit3`, then write zero, to clear FIFO and sticky status.

5. Read `fifospace`. A valid peripheral reports nonzero left and right write space.

6. Load WAV assets from the filesystem, parse RIFF chunks, convert PCM16 mono/stereo to the internal stereo stream, and resample toward `48000` Hz.

7. A worker thread polls `fifospace`; when both channels have room, it writes the left sample to `regs[2]` and right sample to `regs[3]`.

3.7 Audio Pins

Port	Width	Direction	Data
`aud_xck`	1	output	WM8731 master clock, default `12.5 MHz`
`aud_bclk`	1	output	Audio bit clock, default `3.125 MHz`
`aud_daclrck`	1	output	DAC left/right word-select clock
`aud_adclrck`	1	output	ADC left/right word-select clock, synchronized to DAC LRCK
`aud_dacdat`	1	output	Left-justified serial DAC sample bitstream
`aud_adcdat`	1	input	ADC serial data input; present but not consumed by current software
`fpga_i2c_sclk`	1	bidir	Open-drain style I2C SCL for codec initialization
`fpga_i2c_sdat`	1	bidir	Open-drain style I2C SDA for codec initialization and ACK sampling
`codec_init_done`	1	output	Codec initialization completed; also routed to `LEDR[0]`
`codec_init_error`	1	output	Codec initialization failed; also routed to `LEDR[1]`

3.8 Software Game-State MMIO Encoder

`sw/render_if/fighter_mmio.c` still implements `fighter_mmio_encode()`, and `sw/include/fighter_` still defines a 22-word game-state register layout. This contract is useful for tests and for a possible future hardware sprite engine, but it is not the register map consumed by the current framebuffer VGA IP.

Offset	Register	Software Encoding Meaning
`0x00`	`GAME_STATE`	`bit[1:0]` menu/playing/game-over, `bit8` menu frame, `bit9` game-over ready
`0x04`	`PLAYER1_X`	Player 1 top-left x coordinate
`0x08`	`PLAYER1_Y`	Player 1 top-left y coordinate
`0x0C`	`PLAYER1_STATE`	Player 1 visual state

Offset	Register	Software Encoding Meaning
0x10	PLAYER2_X	Player 2 top-left x coordinate
0x14	PLAYER2_Y	Player 2 top-left y coordinate
0x18	PLAYER2_STATE	Player 2 visual state
0x1C	PLAYER1_HP	Player 1 HP
0x20	PLAYER2_HP	Player 2 HP
0x24	ROUND_TIMER	Remaining seconds
0x28	WINNER	Winner enum
0x2C	PLAYER1_FACING	1 = right, 0 = left
0x30	PLAYER2_FACING	1 = right, 0 = left
0x34	DEBUG_FLAGS	Last attacks and attack phases
0x38	PLAYER1_ATTACK_CMD	Player 1 last attack command
0x3C	PLAYER2_ATTACK_CMD	Player 2 last attack command
0x40	PLAYER1_STATE_FRAME	Frames spent in current visual state
0x44	PLAYER2_STATE_FRAME	Frames spent in current visual state
0x48	PLAYER1_EVENT_FLAGS	Per-frame event pulses
0x4C	PLAYER2_EVENT_FLAGS	Per-frame event pulses
0x50	PLAYER1_COMBAT_RESULT	None, hit, blocked, trade, or whiff
0x54	PLAYER2_COMBAT_RESULT	None, hit, blocked, trade, or whiff

4 Detailed Hardware Design

4.1 Top-Level Hardware Structure

The board-level top module is hw/soc_system_top.sv. The current project-relevant connections are:

- soc_system provides the HPS skeleton and lightweight bridge.
- fighter_audio_0 is exported through the audio conduit and wired to AUD_* and FPGA_I2C_* pins.
- audio_init_done and audio_init_error are mapped to LEDR[0] and LEDR[1].
- fighter_vga_0 is exported through the VGA conduit and wired to VGA_R/G/B, VGA_HS, VGA_VS, VGA_CLK, VGA_BLANK_N, and VGA_SYNC_N.

4.2 Hardware Module Responsibilities

Module	Responsibility
`soc_system_top`	Board-level wrapper. It connects DE1-SoC pins to the generated Platform Designer system, routes VGA/audio conduits, maps codec status to LEDs, and ties unused board outputs to simple quiet values.
`soc_system`	Platform Designer system. It contains the HPS, lightweight HPS-to-FPGA AXI master, interconnect, reset controllers, `fighter_vga_0`, and `fighter_audio_0`.
`soc_system_mm_interconnect_0`	Avalon-MM interconnect generated by Platform Designer. It decodes lightweight bridge addresses and forwards reads/writes to the selected custom IP.
`fighter_vga_renderer`	Custom VGA IP. It exposes geometry/control registers, accepts framebuffer writes, tracks display/write buffers, generates VGA timing, and outputs 8-bit RGB plus sync signals.
`fighter_audio_wm8731`	Custom audio IP. It exposes a 4-word register map, queues left/right samples in FIFOs, initializes the WM8731 by I2C, derives audio clocks, and serializes samples to the DAC.

4.3 Register Versus BRAM Implementation

The design contains three different kinds of state, and they synthesize differently:

- Small control/status values such as `swap_pending`, `display_buffer`, VGA counters, audio FIFO pointers, FIFO counts, and I2C state are normal flip-flop registers.
- Audio sample storage is inferred as BRAM. The Quartus map/fitter reports show `left_fifo_rtl_0` and `right_fifo_rtl_0` as `altsyncram` simple dual-port RAMs, each `128 x 32` bits. Each FIFO uses one M10K block, so audio uses two M10K blocks total.
- The source version of `fighter_vga_renderer.sv` describes two framebuffer RAM banks with `altsyncram`. The current Quartus fit report, `hw/output_files/soc_system.fit.rpt`, reports zero M10Ks under `fighter_vga_renderer` and only two M10Ks total, both under the audio FIFOs. Therefore the current compiled resource report confirms BRAM for audio FIFOs, not a full synthesized on-chip VGA framebuffer. The software/hardware register protocol is still documented above because it is the intended VGA IP contract in the source tree.

4.4 Internal Structure of `fighter_vga_renderer`

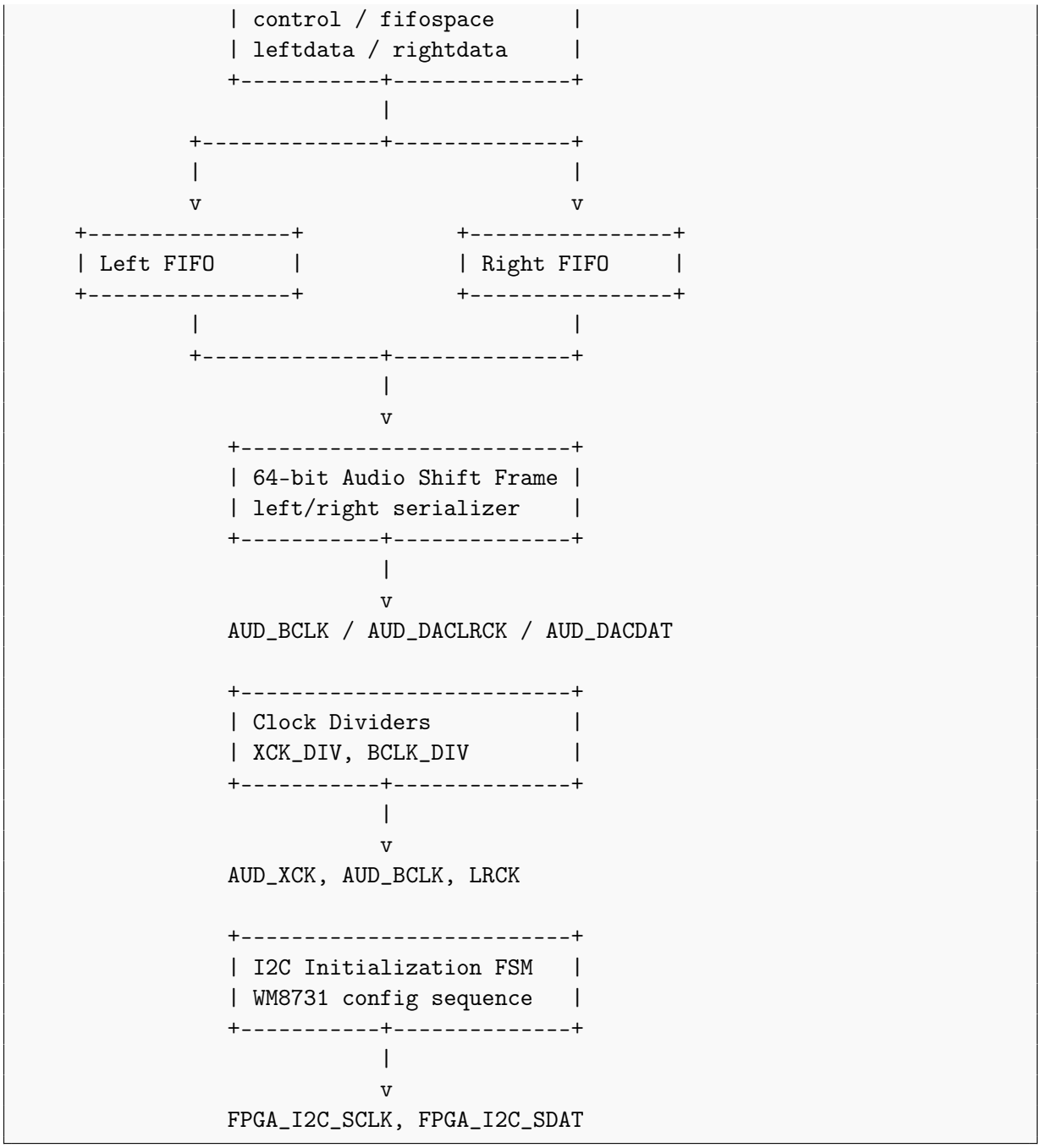
```
Avalon-MM Slave Interface
avs_address / avs_write / avs_writedata / avs_readdata
        |
        +-----------------------------+
        | Register Decode             |
        | control, width, height,     |
```

```
        | stride, ident "VPGA"        |
        +-----------------------------+
        |
        +-----------------------------+
        | Framebuffer Write Logic     |
        | word offset >= 1024         |
        | two RGB565 pixels / word    |
        +-----------------------------+
        |
        v
+-------------------+      +-------------------+
| framebuffer_ram0  |      | framebuffer_ram1  |
| M10K dual-port    |      | M10K dual-port    |
+-------------------+      +-------------------+
        ^                          ^
        |                          |
        +----------+---------------+
                   |
          display_buffer select
                   |
                   v
        Framebuffer Read Address Generator
        640x480 -> 320x240 scaling
        source_x = h_count[9:1]
        source_y = v_count[8:1]
                   |
                   v
        RGB565 Pixel Select
        low halfword / high halfword
                   |
                   v
        RGB Expansion
        R: 5 bits -> 8 bits
        G: 6 bits -> 8 bits
        B: 5 bits -> 8 bits
                   |
                   v
        VGA Timing Generator
        h_count, v_count, hs, vs, blank, pixel clock
                   |
                   v
        VGA_R/G/B, VGA_HS, VGA_VS, VGA_CLK
```

The module uses two framebuffer banks. Software always writes the bank that is not currently displayed. After a full frame is written, software writes `CONTROL_SWAP_REQUEST`; hardware flips the displayed bank at the start of vertical blanking.

4.5 Internal Structure of `fighter_audio_wm8731`

```
                  +--------------------------+
Avalon-MM -----> | Register Map              |
```



4.6 Audio Peripheral Timing and Protocol

`fighter_audio_wm8731` operates in a single 50 MHz clock domain. With default parameters:

- `aud_xck = clk / 4 = 12.5 MHz`
- `aud_bclk = clk / 16 = 3.125 MHz`
- `lrck = clk / 1024 = 48.828125 kHz`

The codec is configured for left-justified format, 16-bit samples, slave mode, DAC playback enabled, and active output.

5 Software Interfaces

5.1 Core Game Interface

`sw/include/fighter_game.h` defines the game data model.

Key enumerations include:

- `fighter_game_state_t`: `MENU / PLAYING / GAME_OVER`
- `fighter_character_id_t`: `RYU / KEN`
- `fighter_visual_state_t`: includes `IDLE`, `WALK`, `JUMP`, `CROUCH`, `GUARD`, `ATTACK`, `HIT`, `BLOCK_STUN`, `KO`, `VICTORY`, and `CROUCH_GUARD`
- `fighter_attack_phase_t`: `NONE`, `STARTUP`, `ACTIVE`, `HIT_CONFIRM`, `BLOCK_CONFIRM`, `RECOVERY`
- `fighter_winner_t` and `fighter_finish_reason_t`

The default configuration uses `640x480`, floor `y = 400`, player size `48x96`, projectile size `28x20`, projectile speed `6`, walk speed `3`, jump velocity `-14`, gravity `1`, max HP `100`, and round duration `99 * 60` frames.

5.2 Input Interface

`sw/include/fighter_input.h` defines the shared input parsing layer. Keyboard and gamepad input are both normalized into `fighter_player_result_t`, so the game logic does not need separate combat code for each physical device.

The default keyboard mapping is:

- `W`: jump
- `A`: move left
- `D`: move right
- `S`: crouch
- `J`: attack
- `K`: guard
- `L`: exit the current match

Attack combinations are decoded in software:

- `J`: normal attack
- `A + J`: fireball
- `D + J`: dragon punch
- `W + J`: jump attack

- `W + D + J`: forward jump attack
- `W + A + J`: back jump attack
- `S + J`: sweep

Gamepad input is handled by `sw/input/fighter_gamepad.c` and `sw/include/fighter_gamepad.h`. The gamepad module opens a Linux input event device, such as `/dev/input/by-id/usb-081f_USB_gamepad-event-joystick`, in nonblocking mode and drains `struct input_event` records each frame.

The gamepad parser handles two event types:

- `EV_ABS`: joystick or directional-pad axes. `ABS_X` is converted into left/right movement and `ABS_Y` is converted into up/down movement.
- `EV_KEY`: button events. Buttons are mapped to attack, guard, fireball, dragon punch, start, and exit.

The gamepad layer stores both the current and previous button states. This allows the software to distinguish held inputs from newly pressed inputs. Held states are used for movement, crouching, jumping, and guarding, while edge-triggered pressed states are used for attacks, menu confirmation, and exiting the match.

For gamepad attacks, dedicated buttons can trigger fireball and dragon punch directly. If the normal attack button is pressed, the current direction is also checked so that direction plus attack can map to sweep, jump attack, fireball, or dragon punch. The result is written into the same `attack_command` field used by the keyboard input path.

5.3 Rendering Interface

`sw/include/fighter_renderer.h` defines three renderer backends:

- `FIGHTER_RENDERER_BACKEND_MMIO`: FPGA VGA framebuffer IP
- `FIGHTER_RENDERER_BACKEND_FRAMEBUFFER`: Linux framebuffer fallback
- `FIGHTER_RENDERER_BACKEND_CONSOLE`: debug console fallback

On Linux, `fighter_renderer_init()` tries the MMIO VGA backend first when framebuffer-style output is preferred. It then falls back to `/dev/fb0`, `/dev/fb1`, or console output. The renderer loads menu PPMs, a background PPM, and character sprites from `game_assets`, with `FIGHTER_ASSET_ROOT` available as an override.

5.4 Audio Interface

`sw/include/fighter_audio.h` defines command-based audio playback.

The current tracks are:

- `MENU_BGM`
- `MENU_CONFIRM`
- `GAME_OVER`

The supported command types are `PLAY_ONCE`, `START_LOOP`, and `STOP_LOOP`. Audio is disabled by default in the demo. It is initialized when `phase1_demo -audio` or `input_demo -audio` is used. Backend selection is `WM8731/MMIO` first, then command-line player fallback, then disabled.

5.5 USB Keyboard Capture Interface

`sw/include/usb_hid_keyboard.h` and `sw/input/usb_hid_keyboard.c` implement device enumeration and polling through `libusb`. The code accepts HID boot keyboards, locates the interrupt IN endpoint, polls 8-byte reports, and supports up to two devices.

6 Verilog Module Interfaces

6.1 `fighter_vga_renderer`

The module definition is in `hw/fighter_vga_renderer.sv`.

Key interface properties:

- clock/reset: `clk_50`, `reset_n`
- Avalon-MM control: `avs_chipselect`, `avs_read`, `avs_write`
- Avalon-MM address/data: `avs_address[15:0]`, `avs_writedata[31:0]`, and `avs_readdata[31:0`
- VGA conduit: `vga_r/g/b[7:0]`, `vga_hs`, `vga_vs`, `vga_clk`, `vga_blank_n`, `vga_sync_n`

Port	Width	Direction	Meaning
`clk_50`	1	input	50 MHz FPGA clock
`reset_n`	1	input	Active-low reset from Platform Designer reset network
`avs_chipselect`	1	input	Avalon slave selected
`avs_read`	1	input	Avalon read strobe
`avs_write`	1	input	Avalon write strobe
`avs_address`	16	input	Word offset for control, geometry, ident, or framebuffer write
`avs_writedata`	32	input	Register write data or packed RGB565 pixels
`avs_readdata`	32	output	Register read data
`vga_r`	8	output	Red channel to VGA DAC
`vga_g`	8	output	Green channel to VGA DAC
`vga_b`	8	output	Blue channel to VGA DAC
`vga_hs`	1	output	VGA horizontal sync
`vga_vs`	1	output	VGA vertical sync
`vga_clk`	1	output	VGA pixel clock
`vga_blank_n`	1	output	VGA blanking indicator
`vga_sync_n`	1	output	VGA sync control, fixed low

6.2 fighter_audio_wm8731

The module definition is in hw/fighter_audio.sv.

Key parameters:

- FIFO_DEPTH = 128
- CLK_HZ = 50000000
- I2C_RATE_HZ = 100000
- XCK_DIV = 4
- BCLK_DIV = 16
- I2C_DEVICE_ADDR = 7'h1A

Key ports include clock/reset, a 4-word Avalon-MM slave, WM8731 audio pins, WM8731 I2C pins, and codec initialization status outputs.

Port	Width	Direction	Meaning
clk	1	input	50 MHz FPGA clock
reset_n	1	input	Active-low reset
avs_chipselect	1	input	Avalon slave selected
avs_read	1	input	Avalon read strobe
avs_write	1	input	Avalon write strobe
avs_address	2	input	Audio register word offset 0..3
avs_writedata	32	input	Control bits or sample word
avs_readdata	32	output	Status or FIFO-space read data
aud_xck	1	output	WM8731 master clock
aud_bclk	1	output	WM8731 bit clock
aud_daclrck	1	output	DAC left/right clock
aud_adclrck	1	output	ADC left/right clock
aud_dacdat	1	output	DAC serial data
aud_adcdat	1	input	ADC serial data, currently unused
fpga_i2c_sclk	1	bidir	Codec I2C clock, open-drain style
fpga_i2c_sdat	1	bidir	Codec I2C data, open-drain style
codec_init_done	1	output	Initialization complete status
codec_init_error	1	output	Initialization failed status

7 Board Resource Use

The current Quartus build report is hw/output_files/soc_system.fit.summary. It was generated by Quartus Prime Lite 21.1 for Cyclone V device 5CSEMA5F31C6.

Resource	Used	Available	Percent
ALMs	1,653	32,070	5%
Registers	1,885	–	–

Resource	Used	Available	Percent
Pins	362	457	79%
Block memory bits	8,192	4,065,280	<1%
RAM blocks / M10Ks	2	397	<1%
DSP blocks	2	87	2%
PLLs	0	6	0%
DLLs	1	4	25%

The fitter hierarchy report attributes the two M10K blocks to `fighter_audio_wm8731`: one for the left FIFO and one for the right FIFO. Each FIFO is `4096` bits, implemented as `128 x 32`. The two DSP blocks are attributed to `fighter_vga_renderer` arithmetic logic. The custom `fighter_audio_wm8731` entity uses approximately `164.4` ALMs, `250` combinational ALUTs, `332` dedicated logic registers, `8192` block memory bits, and `2` M10Ks. The custom `fighter_vga_renderer` entity uses approximately `1100.3` ALMs, `1817` combinational ALUTs, `776` dedicated logic registers, `0` M10Ks in the reported fit, and `2` DSP blocks.

Board-level resources used by the project include:

- HPS DDR3, USB, SD, UART, Ethernet, SPI, and I2C pins through the generated HPS subsystem.
- VGA output pins: 8-bit red, 8-bit green, 8-bit blue, HS, VS, CLK, BLANK_N, and SYNC_N.
- Audio codec pins: `AUD_XCK`, `AUD_BCLK`, `AUD_DACLRCK`, `AUD_ADCLRCK`, `AUD_DACDAT`, and `AUD_ADCDAT`.
- FPGA-side codec I2C pins: `FPGA_I2C_SCLK` and `FPGA_I2C_SDAT`.
- LEDs: `LEDR[0]` shows codec initialization done and `LEDR[1]` shows codec initialization error.

8 Implementation Details and Testing

8.1 Main Loop and Execution Modes

`sw/main_phase1_demo.c` is the main demo program. It supports:

- `-script smoke|ko`
- `-usb`
- `-console`
- `-fb PATH`
- `-audio`
- `-frames N`
- `-realtime`

At runtime, each logic step collects or synthesizes input reports, parses them into `fighter_player_result_t`, updates the game, updates animation state, and processes audio commands. A rendered frame is drawn after at least one logic step.

8.2 Game Rules Already Implemented

The current branch supports:

- left/right movement, jumping, crouching, crouch guard, standing guard
- normal attack, fireball, dragon punch, jump attacks, and sweep
- fireball projectile spawning, movement, collision, mutual cancellation, hit, and block handling
- attack startup, active, hit-confirm, block-confirm, and recovery phases
- automatic facing updates and grounded overlap resolution
- hit damage, guard chip damage, hit stun, block stun, KO, double KO, timeout, exit, menu, game over, and victory/KO visual states

8.3 Automated Test Coverage

`sw/tests/test_phase1.c` covers menu entry, match start, exiting to menu, KO and game-over transitions, restart after game-over gating, timeout draw resolution, MMIO game-state encoding, hit-confirm behavior, block stun, airborne cross-up facing reversal, grounded overlap resolution, projectile behavior, and animation-related state transitions.

8.4 Build Outputs

`sw/Makefile` currently builds:

- `phase1_demo`
- `phase1_test`
- `audio_demo`
- `audio_probe`
- `vga_probe`
- `input_demo`, only when `libusb-1.0` is installed

Recommended validation commands:

```
cd sw
make
./phase1_test
./vga_probe
./audio_probe
./phase1_demo --script smoke --console
```

```
./phase1_demo --script ko --console
./phase1_demo --script smoke --audio
make gamepad_probe
./gamepad_probe /dev/input/by-id/usb-081f_USB_gamepad-event-joystick
```

In the current `sw/Makefile`, `gamepad_probe` has a build rule, but it is not included in the default `all` target. Therefore it should be built directly with `make gamepad_probe`.

8.5 Testing Methodology

In addition to automated gameplay tests, the final validation process includes hardware-facing probe tests and asset-format checks:

- **VGA probe test:** `vga_probe` verifies that the VGA IP can be mapped through `/dev/mem`, returns the expected identifier, and accepts framebuffer writes.
- **Audio probe test:** `audio_probe` verifies the WM8731 MMIO register path, FIFO-space reads, and sample-write path before full game audio playback.
- **Gamepad probe test:** `main_gamepad_probe.c` prints Linux input event type, code, value, and symbolic name. This validates event codes, axes, and button mappings before integrating the controller with gameplay.
- **Asset conversion test:** run `scripts/convert_ppm_to_rgb565.py` on a known `P6` PPM and verify the `.rgb565` header fields, signature `R565`, image dimensions, pixel format ID, pixel data size, and total output size.

8.6 Current Risks and Limitations

- There is no Linux kernel driver yet; VGA and audio hardware are mapped directly through `/dev/mem`.
- The current FPGA VGA renderer is framebuffer-based. It does not yet implement an independent hardware sprite engine that consumes the 22-word game-state MMIO encoding.
- Audio is disabled unless requested with `-audio`.
- The code relies on asset files under `game_assets`; deployment to the board must preserve the expected asset layout or set `FIGHTER_ASSET_ROOT`.
- `input_demo` depends on `libusb-1.0`; the Makefile only builds it when pkg-config can find libusb.

9 Conclusion and Future Work

The current repository contains a working HPS fighting-game prototype and two custom FPGA output peripherals: VGA framebuffer output and WM8731 audio output. The most important architectural distinction is that rendering is currently split as "HPS software draws pixels, FPGA scans them out," not "HPS writes game objects, FPGA draws sprites."

The most direct future roadmap is:

1. keep the current framebuffer VGA path as the stable display backend
2. add a kernel driver or UIO-style access layer to replace raw `/dev/mem`
3. optionally add a true FPGA sprite/UI renderer that consumes the existing 22-word `fighter_mmio_encode()` contract
4. expand board-level validation for USB input, VGA, audio, and asset loading together

A Reference Source Files

A.1 Main Software Source Files

A.1.1 sw/audio/fighter_audio.c

```
#include "fighter_audio.h"

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <pthread.h>
#include <signal.h>
#include <stdarg.h>
#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

enum {
  FIGHTER_PLAYER_KIND_NONE = 0,
  FIGHTER_PLAYER_KIND_APLAY,
  FIGHTER_PLAYER_KIND_FFPLAY
};

enum {
  FIGHTER_AUDIO_TRACK_STORAGE_COUNT = FIGHTER_AUDIO_TRACK_GAME_OVER + 1
};

enum {
  FIGHTER_AUDIO_MMIO_TARGET_RATE = 48000,
  FIGHTER_AUDIO_MMIO_CONTROL_CLEAR_READ = 1 << 2,
  FIGHTER_AUDIO_MMIO_CONTROL_CLEAR_WRITE = 1 << 3
};

/* Audio MMIO register layout follows the DE1-SoC/Lab 3 audio core contract. */
enum {
  FIGHTER_AUDIO_MMIO_REG_CONTROL = 0,
  FIGHTER_AUDIO_MMIO_REG_FIFOSPACE = 1,
  FIGHTER_AUDIO_MMIO_REG_LEFTDATA = 2,
  FIGHTER_AUDIO_MMIO_REG_RIGHTDATA = 3,
  FIGHTER_AUDIO_MMIO_REG_COUNT = 4
};

typedef struct {
  int16_t *samples;
  size_t frame_count;
} fighter_audio_clip_t;

typedef struct {
  pthread_t thread;
  pthread_mutex_t mutex;
  int mutex_initialized;
  int thread_started;
  int stop_requested;
  int mem_fd;
  void *bridge_map;
  size_t bridge_map_length;
  volatile uint32_t *bridge_reset_reg;
  void *audio_map;
  size_t audio_map_length;
  volatile uint32_t *audio_regs;
  fighter_audio_clip_t clips[FIGHTER_AUDIO_TRACK_STORAGE_COUNT];
  fighter_audio_track_t current_track;
  fighter_audio_track_t loop_track;
```

```
  size_t current_frame;
  int playing;
} fighter_audio_mmio_state_t;

static const off_t k_fighter_audio_default_bridge_reset_addr = (off_t)0xFFD0501C;
static const off_t k_fighter_audio_default_mmio_addr = (off_t)0xFF200000;

static void fighter_audio_set_status_detail(fighter_audio_context_t *context,
                                            const char *fmt,
                                            ...) {
  va_list args;

  if (!context || !fmt) {
    return;
  }

  va_start(args, fmt);
  vsnprintf(context->status_detail, sizeof(context->status_detail), fmt, args);
  va_end(args);
}

static const char *fighter_find_in_path(const char *name) {
  static char resolved_path[512];
  const char *path_env;
  const char *segment;

  if (!name || strchr(name, '/') != NULL) {
    return NULL;
  }

  path_env = getenv("PATH");
  if (!path_env) {
    return NULL;
  }

  segment = path_env;
  while (*segment != '\0') {
    const char *separator = strchr(segment, ':');
    size_t prefix_len =
        separator ? (size_t)(separator - segment) : strlen(segment);

    if (prefix_len + 1 + strlen(name) + 1 < sizeof(resolved_path)) {
      memcpy(resolved_path, segment, prefix_len);
      resolved_path[prefix_len] = '/';
      strcpy(resolved_path + prefix_len + 1, name);
      if (access(resolved_path, X_OK) == 0) {
        return resolved_path;
      }
    }

    if (!separator) {
      break;
    }
    segment = separator + 1;
  }

  return NULL;
}

static void fighter_audio_reap_children(void) {
  while (waitpid(-1, NULL, WNOHANG) > 0) {
  }
}

static int fighter_audio_shell_quote(const char *src, char *dst, size_t size) {
  size_t used = 0;

  if (!src || !dst || size < 3) {
    return -1;
  }

  dst[used++] = '\'';
  while (*src != '\0') {
```

```
    if (*src == '\'') {
      if (used + 4 >= size) {
        return -1;
      }
      memcpy(dst + used, "'\\''", 4);
      used += 4;
    } else {
      if (used + 1 >= size) {
        return -1;
      }
      dst[used++] = *src;
    }
    ++src;
  }

  if (used + 2 > size) {
    return -1;
  }
  dst[used++] = '\'';
  dst[used] = '\0';
  return 0;
}

static int fighter_audio_spawn_shell(const char *command) {
  pid_t pid;

  if (!command) {
    return -1;
  }

  pid = fork();
  if (pid < 0) {
    return -1;
  }

  if (pid == 0) {
    setsid();
    execl("/bin/sh", "sh", "-c", command, (char *)NULL);
    _exit(127);
  }

  return (int)pid;
}

static int fighter_audio_copy_string(const char *src, char *dst, size_t dst_size) {
  if (!src || !dst || dst_size == 0) {
    return -1;
  }

  if (strlen(src) + 1 > dst_size) {
    return -1;
  }

  memcpy(dst, src, strlen(src) + 1);
  return 0;
}

static int fighter_audio_parse_env_address(const char *env_name,
                                           off_t default_value,
                                           off_t *value_out) {
  const char *text;
  char *end;
  unsigned long long parsed;

  if (!env_name || !value_out) {
    return -1;
  }

  text = getenv(env_name);
  if (!text || text[0] == '\0') {
    *value_out = default_value;
    return 0;
  }
```

```
212
213   errno = 0;
214   parsed = strtoull(text, &end, 0);
215   if (errno != 0 || end == text || !end || *end != '\0') {
216     return -1;
217   }
218
219   *value_out = (off_t)parsed;
220   return 0;
221 }
222
223 static int fighter_audio_detect_aplay_device(char *buffer, size_t buffer_size) {
224   const char *override;
225   FILE *stream;
226   char line[256];
227   int card;
228   int device;
229
230   if (!buffer || buffer_size == 0 || !fighter_find_in_path("aplay")) {
231     return -1;
232   }
233
234   override = getenv("FIGHTER_AUDIO_DEVICE");
235   if (override && override[0] != '\0') {
236     return fighter_audio_copy_string(override, buffer, buffer_size);
237   }
238
239   stream = popen("aplay -l 2>/dev/null", "r");
240   if (!stream) {
241     return -1;
242   }
243
244   while (fgets(line, sizeof(line), stream)) {
245     if (sscanf(line, "card %d: %*[^,], device %d:", &card, &device) == 2) {
246       snprintf(buffer, buffer_size, "plughw:%d,%d", card, device);
247       (void)pclose(stream);
248       return 0;
249     }
250   }
251
252   (void)pclose(stream);
253   return -1;
254 }
255
256 static void fighter_audio_build_once_command(const fighter_audio_context_t *context,
257                                              const char *quoted_path,
258                                              char *buffer,
259                                              size_t buffer_size) {
260   char quoted_device[128];
261
262   if (!context) {
263     buffer[0] = '\0';
264     return;
265   }
266
267   switch (context->player_kind) {
268     case FIGHTER_PLAYER_KIND_APLAY:
269       if (fighter_audio_shell_quote(context->aplay_device, quoted_device,
270                                     sizeof(quoted_device)) != 0) {
271         buffer[0] = '\0';
272         break;
273       }
274       snprintf(buffer, buffer_size, "aplay -q -D %s %s >/dev/null 2>&1",
275                quoted_device, quoted_path);
276       break;
277     case FIGHTER_PLAYER_KIND_FFPLAY:
278       snprintf(buffer, buffer_size,
279                "ffplay -nodisp -autoexit -loglevel quiet %s >/dev/null 2>&1",
280                quoted_path);
281       break;
282     default:
283       buffer[0] = '\0';
284       break;
```

```
  }
}

static void fighter_audio_build_loop_command(const fighter_audio_context_t *context,
                                             const char *quoted_path,
                                             char *buffer,
                                             size_t buffer_size) {
  char quoted_device[128];

  if (!context) {
    buffer[0] = '\0';
    return;
  }

  switch (context->player_kind) {
    case FIGHTER_PLAYER_KIND_APLAY:
      if (fighter_audio_shell_quote(context->aplay_device, quoted_device,
                                    sizeof(quoted_device)) != 0) {
        buffer[0] = '\0';
        break;
      }
      snprintf(buffer, buffer_size,
               "while aplay -q -D %s %s >/dev/null 2>&1; do :; done",
               quoted_device, quoted_path);
      break;
    case FIGHTER_PLAYER_KIND_FFPLAY:
      snprintf(buffer, buffer_size,
               "ffplay -nodisp -autoexit -loglevel quiet -loop 0 %s "
               ">/dev/null 2>&1",
               quoted_path);
      break;
    default:
      buffer[0] = '\0';
      break;
  }
}

static int fighter_audio_prepare_command_backend(fighter_audio_context_t *context) {
  if (!context) {
    return -1;
  }

  context->aplay_device[0] = '\0';
  if (fighter_audio_detect_aplay_device(context->aplay_device,
                                        sizeof(context->aplay_device)) == 0) {
    context->player_kind = FIGHTER_PLAYER_KIND_APLAY;
    return 0;
  }

  if (fighter_find_in_path("ffplay")) {
    context->player_kind = FIGHTER_PLAYER_KIND_FFPLAY;
    return 0;
  }

  context->player_kind = FIGHTER_PLAYER_KIND_NONE;
  return -1;
}

static void fighter_audio_clip_reset(fighter_audio_clip_t *clip) {
  if (!clip) {
    return;
  }

  free(clip->samples);
  clip->samples = NULL;
  clip->frame_count = 0;
}

static uint16_t fighter_audio_read_le16(const unsigned char *src) {
  return (uint16_t)src[0] | (uint16_t)((uint16_t)src[1] << 8);
}

static uint32_t fighter_audio_read_le32(const unsigned char *src) {
```

```
  return (uint32_t)src[0] | ((uint32_t)src[1] << 8) |
         ((uint32_t)src[2] << 16) | ((uint32_t)src[3] << 24);
}

static int fighter_audio_resample_pcm16(fighter_audio_clip_t *clip,
                                        const int16_t *src_samples,
                                        size_t src_frame_count,
                                        uint16_t src_channels,
                                        uint32_t src_rate) {
  int16_t *dst_samples;
  size_t dst_frame_count;
  size_t i;

  if (!clip || !src_samples || src_frame_count == 0 || src_rate == 0 ||
      (src_channels != 1 && src_channels != 2)) {
    return -1;
  }

  dst_frame_count =
      (size_t)(((uint64_t)src_frame_count * FIGHTER_AUDIO_MMIO_TARGET_RATE +
                src_rate - 1) /
               src_rate);
  if (dst_frame_count == 0) {
    dst_frame_count = 1;
  }

  dst_samples =
      (int16_t *)malloc(dst_frame_count * 2U * sizeof(int16_t));
  if (!dst_samples) {
    return -1;
  }

  for (i = 0; i < dst_frame_count; ++i) {
    uint64_t src_position_num = (uint64_t)i * src_rate;
    size_t src_index = (size_t)(src_position_num / FIGHTER_AUDIO_MMIO_TARGET_RATE);
    uint32_t frac =
        (uint32_t)(src_position_num % FIGHTER_AUDIO_MMIO_TARGET_RATE);
    size_t next_index;
    int channel;

    if (src_index >= src_frame_count) {
      src_index = src_frame_count - 1;
    }
    next_index =
        src_index + 1 < src_frame_count ? src_index + 1 : src_index;

    for (channel = 0; channel < 2; ++channel) {
      int src_channel = src_channels == 1 ? 0 : channel;
      int32_t sample_a =
          src_samples[src_index * src_channels + (size_t)src_channel];
      int32_t sample_b =
          src_samples[next_index * src_channels + (size_t)src_channel];
      int32_t blended =
          sample_a +
          (int32_t)(((int64_t)(sample_b - sample_a) * frac) /
                    FIGHTER_AUDIO_MMIO_TARGET_RATE);
      dst_samples[i * 2U + (size_t)channel] = (int16_t)blended;
    }
  }

  clip->samples = dst_samples;
  clip->frame_count = dst_frame_count;
  return 0;
}

static int fighter_audio_clip_load_wav(fighter_audio_clip_t *clip,
                                       const char *path) {
  FILE *stream;
  unsigned char header[12];
  unsigned char chunk_header[8];
  unsigned char *data_bytes = NULL;
  size_t data_size = 0;
  uint16_t audio_format = 0;
```

```
  uint16_t channel_count = 0;
  uint16_t bits_per_sample = 0;
  uint32_t sample_rate = 0;
  int have_format = 0;
  int have_data = 0;
  int result = -1;

  if (!clip || !path) {
    return -1;
  }

  stream = fopen(path, "rb");
  if (!stream) {
    return -1;
  }

  if (fread(header, 1, sizeof(header), stream) != sizeof(header) ||
      memcmp(header, "RIFF", 4) != 0 || memcmp(header + 8, "WAVE", 4) != 0) {
    fclose(stream);
    return -1;
  }

  while (fread(chunk_header, 1, sizeof(chunk_header), stream) ==
         sizeof(chunk_header)) {
    uint32_t chunk_size = fighter_audio_read_le32(chunk_header + 4);

    if (memcmp(chunk_header, "fmt ", 4) == 0) {
      unsigned char *fmt_data;

      if (chunk_size < 16) {
        break;
      }
      fmt_data = (unsigned char *)malloc(chunk_size);
      if (!fmt_data) {
        break;
      }
      if (fread(fmt_data, 1, chunk_size, stream) != chunk_size) {
        free(fmt_data);
        break;
      }

      audio_format = fighter_audio_read_le16(fmt_data);
      channel_count = fighter_audio_read_le16(fmt_data + 2);
      sample_rate = fighter_audio_read_le32(fmt_data + 4);
      bits_per_sample = fighter_audio_read_le16(fmt_data + 14);
      have_format = 1;
      free(fmt_data);
    } else if (memcmp(chunk_header, "data", 4) == 0) {
      data_bytes = (unsigned char *)malloc(chunk_size);
      if (!data_bytes) {
        break;
      }
      if (fread(data_bytes, 1, chunk_size, stream) != chunk_size) {
        free(data_bytes);
        data_bytes = NULL;
        break;
      }
      data_size = chunk_size;
      have_data = 1;
    } else {
      if (fseek(stream, (long)chunk_size, SEEK_CUR) != 0) {
        break;
      }
    }

    if ((chunk_size & 1U) != 0U) {
      if (fseek(stream, 1L, SEEK_CUR) != 0) {
        break;
      }
    }

    if (have_format && have_data) {
      size_t frame_count;
```

```
504
505       if (audio_format != 1 || bits_per_sample != 16 ||
506           (channel_count != 1 && channel_count != 2) || sample_rate == 0) {
507         break;
508       }
509
510       frame_count = data_size / ((size_t)channel_count * sizeof(int16_t));
511       if (frame_count == 0) {
512         break;
513       }
514
515       fighter_audio_clip_reset(clip);
516       result = fighter_audio_resample_pcm16(
517           clip, (const int16_t *)data_bytes, frame_count, channel_count,
518           sample_rate);
519       break;
520     }
521   }
522
523   free(data_bytes);
524   fclose(stream);
525   return result;
526 }
527
528 static int fighter_audio_map_physical(int mem_fd,
529                                       off_t physical_addr,
530                                       size_t span,
531                                       void **map_base,
532                                       size_t *map_length,
533                                       volatile uint32_t **register_base) {
534   long page_size;
535   off_t page_base;
536   off_t page_offset;
537   size_t length;
538   void *mapped;
539
540   if (mem_fd < 0 || !map_base || !map_length || !register_base || span == 0) {
541     return -1;
542   }
543
544   page_size = sysconf(_SC_PAGESIZE);
545   if (page_size <= 0) {
546     return -1;
547   }
548
549   page_base = physical_addr & ~((off_t)page_size - 1);
550   page_offset = physical_addr - page_base;
551   length = (size_t)page_offset + span;
552   length = (length + (size_t)page_size - 1U) & ~((size_t)page_size - 1U);
553
554   mapped = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd,
555                 page_base);
556   if (mapped == MAP_FAILED) {
557     return -1;
558   }
559
560   *map_base = mapped;
561   *map_length = length;
562   *register_base =
563       (volatile uint32_t *)((unsigned char *)mapped + (size_t)page_offset);
564   return 0;
565 }
566
567 static void fighter_audio_mmio_unmap_region(void **map_base, size_t *map_length) {
568   if (!map_base || !map_length || !*map_base || *map_length == 0) {
569     return;
570   }
571
572   munmap(*map_base, *map_length);
573   *map_base = NULL;
574   *map_length = 0;
575 }
576
```

```
static void fighter_audio_mmio_reset_clips(fighter_audio_mmio_state_t *state) {
  int i;

  if (!state) {
    return;
  }

  for (i = 0; i < FIGHTER_AUDIO_TRACK_STORAGE_COUNT; ++i) {
    fighter_audio_clip_reset(&state->clips[i]);
  }
}

static int fighter_audio_mmio_load_clips(fighter_audio_mmio_state_t *state) {
  int track;

  if (!state) {
    return -1;
  }

  for (track = 1; track < FIGHTER_AUDIO_TRACK_STORAGE_COUNT; ++track) {
    if (fighter_audio_clip_load_wav(&state->clips[track],
                                    fighter_audio_track_path(
                                        (fighter_audio_track_t)track)) != 0) {
      fighter_audio_mmio_reset_clips(state);
      return -1;
    }
  }

  return 0;
}

static void fighter_audio_mmio_clear_fifos(fighter_audio_mmio_state_t *state) {
  if (!state || !state->audio_regs) {
    return;
  }

  state->audio_regs[FIGHTER_AUDIO_MMIO_REG_CONTROL] =
      FIGHTER_AUDIO_MMIO_CONTROL_CLEAR_READ |
      FIGHTER_AUDIO_MMIO_CONTROL_CLEAR_WRITE;
  state->audio_regs[FIGHTER_AUDIO_MMIO_REG_CONTROL] = 0;
}

static int fighter_audio_mmio_probe(fighter_audio_mmio_state_t *state,
                                    uint32_t *fifospace_out) {
  uint32_t fifospace;
  uint32_t write_space_left;
  uint32_t write_space_right;

  if (!state || !state->audio_regs) {
    return -1;
  }

  fighter_audio_mmio_clear_fifos(state);
  fifospace = state->audio_regs[FIGHTER_AUDIO_MMIO_REG_FIFOSPACE];
  if (fifospace_out) {
    *fifospace_out = fifospace;
  }
  write_space_left = (fifospace >> 24) & 0xFFU;
  write_space_right = (fifospace >> 16) & 0xFFU;

  if (write_space_left == 0 || write_space_left > 128 ||
      write_space_right == 0 || write_space_right > 128) {
    return -1;
  }

  return 0;
}

static int fighter_audio_mmio_enable_bridges(fighter_audio_mmio_state_t *state) {
  uint32_t value;

  if (!state || !state->bridge_reset_reg) {
    return -1;
```

```
  }

  value = *state->bridge_reset_reg;
  value &= ~0x3U;
  *state->bridge_reset_reg = value;
  return 0;
}

static int fighter_audio_track_is_valid(fighter_audio_track_t track) {
  return track > FIGHTER_AUDIO_TRACK_NONE &&
         track <= FIGHTER_AUDIO_TRACK_GAME_OVER;
}

static void fighter_audio_mmio_set_track_locked(fighter_audio_mmio_state_t *state,
                                                fighter_audio_track_t track,
                                                int loop_enabled) {
  if (!state) {
    return;
  }

  if (!fighter_audio_track_is_valid(track) ||
      !state->clips[track].samples || state->clips[track].frame_count == 0) {
    state->current_track = FIGHTER_AUDIO_TRACK_NONE;
    state->loop_track = FIGHTER_AUDIO_TRACK_NONE;
    state->current_frame = 0;
    state->playing = 0;
    return;
  }

  state->current_track = track;
  state->loop_track = loop_enabled ? track : FIGHTER_AUDIO_TRACK_NONE;
  state->current_frame = 0;
  state->playing = 1;
}

static void fighter_audio_mmio_stop_locked(fighter_audio_mmio_state_t *state) {
  if (!state) {
    return;
  }

  state->current_track = FIGHTER_AUDIO_TRACK_NONE;
  state->loop_track = FIGHTER_AUDIO_TRACK_NONE;
  state->current_frame = 0;
  state->playing = 0;
}

static void fighter_audio_mmio_fill_fifo_locked(fighter_audio_mmio_state_t *state) {
  fighter_audio_clip_t *clip;
  uint32_t fifospace;
  size_t writable_frames;
  size_t frame_index;

  if (!state || !state->audio_regs || !state->playing ||
      !fighter_audio_track_is_valid(state->current_track)) {
    return;
  }

  clip = &state->clips[state->current_track];
  if (!clip->samples || clip->frame_count == 0) {
    fighter_audio_mmio_stop_locked(state);
    return;
  }

  fifospace = state->audio_regs[FIGHTER_AUDIO_MMIO_REG_FIFOSPACE];
  writable_frames = (size_t)((fifospace >> 24) & 0xFFU);
  if (((fifospace >> 16) & 0xFFU) < writable_frames) {
    writable_frames = (size_t)((fifospace >> 16) & 0xFFU);
  }

  for (frame_index = 0; frame_index < writable_frames; ++frame_index) {
    int16_t left_sample;
    int16_t right_sample;

```

```
723     if (state->current_frame >= clip->frame_count) {
724       if (state->loop_track == state->current_track) {
725         state->current_frame = 0;
726       } else {
727         fighter_audio_mmio_stop_locked(state);
728         break;
729       }
730     }
731
732     left_sample = clip->samples[state->current_frame * 2U];
733     right_sample = clip->samples[state->current_frame * 2U + 1U];
734
735
736     state->audio_regs[FIGHTER_AUDIO_MMIO_REG_LEFTDATA] =
737       ((uint32_t)(uint16_t)left_sample) << 16;
738     state->audio_regs[FIGHTER_AUDIO_MMIO_REG_RIGHTDATA] =
739        ((uint32_t)(uint16_t)right_sample) << 16;
740     state->current_frame++;
741   }
742 }
743
744 static void *fighter_audio_mmio_thread_main(void *opaque) {
745   fighter_audio_mmio_state_t *state = (fighter_audio_mmio_state_t *)opaque;
746   struct timespec delay;
747
748   delay.tv_sec = 0;
749   delay.tv_nsec = 1000000L;
750
751   while (1) {
752     pthread_mutex_lock(&state->mutex);
753     if (state->stop_requested) {
754       pthread_mutex_unlock(&state->mutex);
755       break;
756     }
757     fighter_audio_mmio_fill_fifo_locked(state);
758     pthread_mutex_unlock(&state->mutex);
759     nanosleep(&delay, NULL);
760   }
761
762   return NULL;
763 }
764
765 static void fighter_audio_mmio_destroy(fighter_audio_mmio_state_t *state) {
766   if (!state) {
767     return;
768   }
769
770   if (state->thread_started) {
771     pthread_mutex_lock(&state->mutex);
772     state->stop_requested = 1;
773     pthread_mutex_unlock(&state->mutex);
774     pthread_join(state->thread, NULL);
775   }
776
777   fighter_audio_mmio_clear_fifos(state);
778   fighter_audio_mmio_reset_clips(state);
779   fighter_audio_mmio_unmap_region(&state->audio_map, &state->audio_map_length);
780   fighter_audio_mmio_unmap_region(&state->bridge_map, &state->bridge_map_length);
781   if (state->mem_fd >= 0) {
782     close(state->mem_fd);
783   }
784   if (state->mutex_initialized) {
785     pthread_mutex_destroy(&state->mutex);
786   }
787   free(state);
788 }
789
790 static int fighter_audio_mmio_init(fighter_audio_context_t *context) {
791   fighter_audio_mmio_state_t *state;
792   off_t bridge_reset_addr;
793   off_t mmio_addr;
794   uint32_t fifospace = 0;
795   int thread_create_result;
```

```
796
797   if (!context) {
798     return -1;
799   }
800
801   state = (fighter_audio_mmio_state_t *)calloc(1, sizeof(*state));
802   if (!state) {
803     return -1;
804   }
805
806   state->mem_fd = -1;
807   if (pthread_mutex_init(&state->mutex, NULL) != 0) {
808     fighter_audio_mmio_destroy(state);
809     return -1;
810   }
811   state->mutex_initialized = 1;
812
813   if (fighter_audio_parse_env_address("FIGHTER_AUDIO_BRIDGE_RESET_ADDR",
814                                       k_fighter_audio_default_bridge_reset_addr,
815                                       &bridge_reset_addr) != 0) {
816     fighter_audio_set_status_detail(
817         context,
818         "WM8731 MMIO: invalid FIGHTER_AUDIO_BRIDGE_RESET_ADDR=%s",
819         getenv("FIGHTER_AUDIO_BRIDGE_RESET_ADDR"));
820     fighter_audio_mmio_destroy(state);
821     return -1;
822   }
823   if (fighter_audio_parse_env_address("FIGHTER_AUDIO_MMIO_ADDR",
824                                       k_fighter_audio_default_mmio_addr,
825                                       &mmio_addr) != 0) {
826     fighter_audio_set_status_detail(
827         context, "WM8731 MMIO: invalid FIGHTER_AUDIO_MMIO_ADDR=%s",
828         getenv("FIGHTER_AUDIO_MMIO_ADDR"));
829     fighter_audio_mmio_destroy(state);
830     return -1;
831   }
832
833   context->bridge_reset_addr = (unsigned long)bridge_reset_addr;
834   context->mmio_addr = (unsigned long)mmio_addr;
835
836   state->mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
837   if (state->mem_fd < 0) {
838     fighter_audio_set_status_detail(context, "WM8731 MMIO: open /dev/mem failed: %s",
839                                     strerror(errno));
840     fighter_audio_mmio_destroy(state);
841     return -1;
842   }
843
844   if (fighter_audio_map_physical(state->mem_fd, bridge_reset_addr,
845                                  sizeof(uint32_t), &state->bridge_map,
846                                  &state->bridge_map_length,
847                                  &state->bridge_reset_reg) != 0) {
848     fighter_audio_set_status_detail(context,
849                                     "WM8731 MMIO: map bridge reset 0x%08lX failed: %s",
850                                     (unsigned long)bridge_reset_addr,
851                                     strerror(errno));
852     fighter_audio_mmio_destroy(state);
853     return -1;
854   }
855   if (fighter_audio_mmio_enable_bridges(state) != 0) {
856     fighter_audio_set_status_detail(context,
857                                     "WM8731 MMIO: failed to enable FPGA bridges");
858     fighter_audio_mmio_destroy(state);
859     return -1;
860   }
861
862   if (fighter_audio_map_physical(state->mem_fd, mmio_addr,
863                                  FIGHTER_AUDIO_MMIO_REG_COUNT *
864                                      sizeof(uint32_t),
865                                  &state->audio_map,
866                                  &state->audio_map_length,
867                                  &state->audio_regs) != 0) {
868     fighter_audio_set_status_detail(context,
```

```
                                    "WM8731 MMIO: map audio core 0x%08lX failed: %s",
                                    (unsigned long)mmio_addr,
                                    strerror(errno));
    fighter_audio_mmio_destroy(state);
    return -1;
  }
  if (fighter_audio_mmio_probe(state, &fifospace) != 0) {
    fighter_audio_set_status_detail(
        context,
        "WM8731 MMIO: audio FIFO probe failed at 0x%08lX (fifospace=0x%08" PRIX32 ")",
        (unsigned long)mmio_addr, fifospace);
    fighter_audio_mmio_destroy(state);
    return -1;
  }

  if (fighter_audio_mmio_load_clips(state) != 0) {
    fighter_audio_set_status_detail(context,
                                    "WM8731 MMIO: failed to load WAV assets");
    fighter_audio_mmio_destroy(state);
    return -1;
  }

  thread_create_result =
      pthread_create(&state->thread, NULL, fighter_audio_mmio_thread_main, state);
  if (thread_create_result != 0) {
    fighter_audio_set_status_detail(context,
                                    "WM8731 MMIO: audio thread start failed: %s",
                                    strerror(thread_create_result));
    fighter_audio_mmio_destroy(state);
    return -1;
  }
  state->thread_started = 1;

  context->backend = FIGHTER_AUDIO_BACKEND_MMIO;
  context->status_detail[0] = '\0';
  context->backend_data = state;
  return 0;
}

static void fighter_audio_stop_loop_command(fighter_audio_context_t *context) {
  pid_t pid;

  if (!context || context->loop_pid <= 0) {
    return;
  }

  pid = (pid_t)context->loop_pid;
  kill(-pid, SIGTERM);
  waitpid(pid, NULL, 0);
  context->loop_pid = 0;
  context->looping_track = FIGHTER_AUDIO_TRACK_NONE;
}

static void fighter_audio_start_loop_command(fighter_audio_context_t *context,
                                             fighter_audio_track_t track) {
  const char *path;
  char quoted_path[512];
  char command[768];
  int pid;

  if (!context || context->backend != FIGHTER_AUDIO_BACKEND_COMMAND) {
    return;
  }

  if (context->looping_track == track && context->loop_pid > 0) {
    return;
  }

  path = fighter_audio_track_path(track);
  if (!path || fighter_audio_shell_quote(path, quoted_path,
                                         sizeof(quoted_path)) != 0) {
    return;
  }
```

```
942
943   fighter_audio_build_loop_command(context, quoted_path, command,
944                                     sizeof(command));
945   if (command[0] == '\0') {
946     return;
947   }
948
949   fighter_audio_stop_loop_command(context);
950   pid = fighter_audio_spawn_shell(command);
951   if (pid > 0) {
952     context->loop_pid = pid;
953     context->looping_track = track;
954   }
955 }
956
957 static void fighter_audio_play_once_command(fighter_audio_context_t *context,
958                                             fighter_audio_track_t track) {
959   const char *path;
960   char quoted_path[512];
961   char command[768];
962
963   if (!context || context->backend != FIGHTER_AUDIO_BACKEND_COMMAND) {
964     return;
965   }
966
967   path = fighter_audio_track_path(track);
968   if (!path || fighter_audio_shell_quote(path, quoted_path,
969                                          sizeof(quoted_path)) != 0) {
970     return;
971   }
972
973   fighter_audio_build_once_command(context, quoted_path, command,
974                                    sizeof(command));
975   if (command[0] == '\0') {
976     return;
977   }
978
979   (void)fighter_audio_spawn_shell(command);
980 }
981
982 static void fighter_audio_stop_loop_mmio(fighter_audio_context_t *context) {
983   fighter_audio_mmio_state_t *state;
984
985   if (!context || context->backend != FIGHTER_AUDIO_BACKEND_MMIO ||
986       !context->backend_data) {
987     return;
988   }
989
990   state = (fighter_audio_mmio_state_t *)context->backend_data;
991   pthread_mutex_lock(&state->mutex);
992   fighter_audio_mmio_stop_locked(state);
993   fighter_audio_mmio_clear_fifos(state);
994   pthread_mutex_unlock(&state->mutex);
995   context->looping_track = FIGHTER_AUDIO_TRACK_NONE;
996 }
997
998 static void fighter_audio_start_loop_mmio(fighter_audio_context_t *context,
999                                           fighter_audio_track_t track) {
1000   fighter_audio_mmio_state_t *state;
1001
1002   if (!context || context->backend != FIGHTER_AUDIO_BACKEND_MMIO ||
1003       !context->backend_data) {
1004     return;
1005   }
1006
1007   state = (fighter_audio_mmio_state_t *)context->backend_data;
1008   pthread_mutex_lock(&state->mutex);
1009   fighter_audio_mmio_set_track_locked(state, track, 1);
1010   fighter_audio_mmio_clear_fifos(state);
1011   fighter_audio_mmio_fill_fifo_locked(state);
1012   pthread_mutex_unlock(&state->mutex);
1013   context->looping_track = track;
1014 }
```

```
1015
1016 static void fighter_audio_play_once_mmio(fighter_audio_context_t *context,
1017                                          fighter_audio_track_t track) {
1018   fighter_audio_mmio_state_t *state;
1019
1020   if (!context || context->backend != FIGHTER_AUDIO_BACKEND_MMIO ||
1021       !context->backend_data) {
1022     return;
1023   }
1024
1025   state = (fighter_audio_mmio_state_t *)context->backend_data;
1026   pthread_mutex_lock(&state->mutex);
1027   fighter_audio_mmio_set_track_locked(state, track, 0);
1028   fighter_audio_mmio_clear_fifos(state);
1029   fighter_audio_mmio_fill_fifo_locked(state);
1030   pthread_mutex_unlock(&state->mutex);
1031   context->looping_track = FIGHTER_AUDIO_TRACK_NONE;
1032 }
1033
1034 static void fighter_audio_start_loop(fighter_audio_context_t *context,
1035                                      fighter_audio_track_t track) {
1036   if (!context) {
1037     return;
1038   }
1039
1040   if (context->backend == FIGHTER_AUDIO_BACKEND_MMIO) {
1041     fighter_audio_start_loop_mmio(context, track);
1042   } else if (context->backend == FIGHTER_AUDIO_BACKEND_COMMAND) {
1043     fighter_audio_start_loop_command(context, track);
1044   }
1045 }
1046
1047 static void fighter_audio_stop_loop(fighter_audio_context_t *context) {
1048   if (!context) {
1049     return;
1050   }
1051
1052   if (context->backend == FIGHTER_AUDIO_BACKEND_MMIO) {
1053     fighter_audio_stop_loop_mmio(context);
1054   } else if (context->backend == FIGHTER_AUDIO_BACKEND_COMMAND) {
1055     fighter_audio_stop_loop_command(context);
1056   }
1057 }
1058
1059 static void fighter_audio_play_once(fighter_audio_context_t *context,
1060                                     fighter_audio_track_t track) {
1061   if (!context) {
1062     return;
1063   }
1064
1065   if (context->backend == FIGHTER_AUDIO_BACKEND_MMIO) {
1066     fighter_audio_play_once_mmio(context, track);
1067   } else if (context->backend == FIGHTER_AUDIO_BACKEND_COMMAND) {
1068     fighter_audio_play_once_command(context, track);
1069   }
1070 }
1071
1072 void fighter_audio_command_list_clear(fighter_audio_command_list_t *list) {
1073   if (!list) {
1074     return;
1075   }
1076
1077   memset(list, 0, sizeof(*list));
1078 }
1079
1080 int fighter_audio_command_list_push(fighter_audio_command_list_t *list,
1081                                     fighter_audio_command_type_t type,
1082                                     fighter_audio_track_t track) {
1083   if (!list || list->count >= FIGHTER_AUDIO_MAX_COMMANDS) {
1084     return -1;
1085   }
1086
1087   list->commands[list->count].type = type;
```

```
  list->commands[list->count].track = track;
  list->count++;
  return 0;
}

const char *fighter_audio_track_name(fighter_audio_track_t track) {
  switch (track) {
    case FIGHTER_AUDIO_TRACK_MENU_BGM:
      return "menu_bgm";
    case FIGHTER_AUDIO_TRACK_MENU_CONFIRM:
      return "menu_confirm";
    case FIGHTER_AUDIO_TRACK_GAME_OVER:
      return "game_over";
    default:
      return "none";
  }
}

const char *fighter_audio_track_path(fighter_audio_track_t track) {
  switch (track) {
    case FIGHTER_AUDIO_TRACK_MENU_BGM:
      return "../game_assets/sound effects/Title.wav";
    case FIGHTER_AUDIO_TRACK_MENU_CONFIRM:
      return "../game_assets/sound effects/Credit.wav";
    case FIGHTER_AUDIO_TRACK_GAME_OVER:
      return "../game_assets/sound effects/Game Over.wav";
    default:
      return NULL;
  }
}

const char *fighter_audio_backend_name(const fighter_audio_context_t *context) {
  static char description[128];

  if (!context || context->backend == FIGHTER_AUDIO_BACKEND_DISABLED) {
    if (context && context->status_detail[0] != '\0') {
      snprintf(description, sizeof(description), "disabled (%s)",
               context->status_detail);
      return description;
    }
    return "disabled";
  }

  switch (context->backend) {
    case FIGHTER_AUDIO_BACKEND_MMIO:
      snprintf(description, sizeof(description), "wm8731-mmio (0x%08lX)",
               context->mmio_addr != 0 ? context->mmio_addr
                                       : (unsigned long)k_fighter_audio_default_mmio_addr);
      return description;
    case FIGHTER_AUDIO_BACKEND_COMMAND:
      switch (context->player_kind) {
        case FIGHTER_PLAYER_KIND_APLAY:
          if (context->aplay_device[0] != '\0') {
            snprintf(description, sizeof(description), "aplay (%s)",
                     context->aplay_device);
            return description;
          }
          return "aplay";
        case FIGHTER_PLAYER_KIND_FFPLAY:
          return "ffplay";
        case FIGHTER_PLAYER_KIND_NONE:
        default:
          return "command";
      }
    case FIGHTER_AUDIO_BACKEND_DISABLED:
    default:
      return "disabled";
  }
}

void fighter_audio_options_init(fighter_audio_options_t *options) {
  if (!options) {
    return;
```

```
  }

  memset(options, 0, sizeof(*options));
}

int fighter_audio_init(fighter_audio_context_t *context,
                       const fighter_audio_options_t *options) {
  int enable_command_audio;

  if (!context) {
    return -1;
  }

  memset(context, 0, sizeof(*context));
  context->backend = FIGHTER_AUDIO_BACKEND_DISABLED;
  context->mmio_addr = (unsigned long)k_fighter_audio_default_mmio_addr;
  context->bridge_reset_addr =
      (unsigned long)k_fighter_audio_default_bridge_reset_addr;

  enable_command_audio = options && options->enable_command_audio;
  if (!enable_command_audio) {
    return 0;
  }

  if (fighter_audio_mmio_init(context) == 0) {
    return 0;
  }

  if (fighter_audio_prepare_command_backend(context) == 0 &&
      context->player_kind != FIGHTER_PLAYER_KIND_NONE) {
    context->backend = FIGHTER_AUDIO_BACKEND_COMMAND;
    context->status_detail[0] = '\0';
  } else {
    if (context->status_detail[0] != '\0') {
      fprintf(stderr, "audio disabled: %s\n", context->status_detail);
    } else {
      fighter_audio_set_status_detail(
          context, "no usable WM8731/MMIO or command backend found");
      fprintf(stderr, "audio disabled: %s\n", context->status_detail);
    }
  }

  return 0;
}

void fighter_audio_close(fighter_audio_context_t *context) {
  if (!context) {
    return;
  }

  if (context->backend == FIGHTER_AUDIO_BACKEND_MMIO &&
      context->backend_data) {
    fighter_audio_mmio_destroy(
        (fighter_audio_mmio_state_t *)context->backend_data);
    context->backend_data = NULL;
  } else {
    fighter_audio_stop_loop_command(context);
    fighter_audio_reap_children();
  }

  context->backend = FIGHTER_AUDIO_BACKEND_DISABLED;
  context->looping_track = FIGHTER_AUDIO_TRACK_NONE;
}

void fighter_audio_process_commands(fighter_audio_context_t *context,
                                    const fighter_audio_command_list_t *commands) {
  size_t i;

  fighter_audio_reap_children();

  if (!context || !commands) {
    return;
  }
```

```

  if (context->backend == FIGHTER_AUDIO_BACKEND_DISABLED &&
      !context->backend_logged) {
    context->backend_logged = 1;
  }

  for (i = 0; i < commands->count; ++i) {
    const fighter_audio_command_t *command = &commands->commands[i];
    switch (command->type) {
      case FIGHTER_AUDIO_COMMAND_PLAY_ONCE:
        fighter_audio_play_once(context, command->track);
        break;
      case FIGHTER_AUDIO_COMMAND_START_LOOP:
        fighter_audio_start_loop(context, command->track);
        break;
      case FIGHTER_AUDIO_COMMAND_STOP_LOOP:
        fighter_audio_stop_loop(context);
        break;
      case FIGHTER_AUDIO_COMMAND_NONE:
      default:
        break;
    }
  }
}
```

A.1.2 sw/game/fighter_game.c

```
#include "fighter_game.h"

#include <string.h>

typedef struct {
  int reach;
  int damage;
  int startup_frames;
  int active_frames;
  int recovery_frames;
  int hit_confirm_frames;
  int block_confirm_frames;
  int hit_stun_frames;
  int block_stun_frames;
} fighter_attack_profile_t;

static int fighter_clamp_int(int value, int min_value, int max_value) {
  if (value < min_value) {
    return min_value;
  }
  if (value > max_value) {
    return max_value;
  }
  return value;
}

static int fighter_rects_overlap(int lhs_x,
                                 int lhs_y,
                                 int lhs_w,
                                 int lhs_h,
                                 int rhs_x,
                                 int rhs_y,
                                 int rhs_w,
                                 int rhs_h) {
  return lhs_x < rhs_x + rhs_w && lhs_x + lhs_w > rhs_x &&
         lhs_y < rhs_y + rhs_h && lhs_y + lhs_h > rhs_y;
}


static void fighter_projectile_reset(fighter_projectile_state_t *projectile) {
  if (!projectile) {
    return;
  }
```

```
44
45    memset(projectile, 0, sizeof(*projectile));
46    projectile->owner_index = -1;
47  }
48
49
50  static int fighter_player_ground_y(const fighter_game_t *game) {
51    return game->config.floor_y - game->config.player_height;
52  }
53
54
55  static int fighter_player_center_x(const fighter_game_t *game,
56                                     const fighter_player_state_t *player) {
57    return player->x + game->config.player_width / 2;
58  }
59
60
61  static int fighter_player_is_airborne(const fighter_game_t *game,
62                                        const fighter_player_state_t *player) {
63    return player->y < fighter_player_ground_y(game) || player->vy != 0;
64  }
65
66  static int fighter_player_vertical_overlap(const fighter_game_t *game,
67                                             const fighter_player_state_t *lhs,
68                                             const fighter_player_state_t *rhs) {
69    int top;
70    int bottom;
71    int lhs_bottom;
72    int rhs_bottom;
73
74    if (!game || !lhs || !rhs) {
75      return 0;
76    }
77
78    top = lhs->y > rhs->y ? lhs->y : rhs->y;
79    lhs_bottom = lhs->y + game->config.player_height;
80    rhs_bottom = rhs->y + game->config.player_height;
81    bottom = lhs_bottom < rhs_bottom ? lhs_bottom : rhs_bottom;
82    return bottom - top;
83  }
84
85  static fighter_attack_profile_t fighter_attack_profile(
86      fighter_attack_command_t attack) {
87    switch (attack) {
88      case FIGHTER_ATTACK_NORMAL:
89        return (fighter_attack_profile_t){48, 12, 3, 2, 15, 2, 2, 8, 3};
90      case FIGHTER_ATTACK_FIREBALL:
91        return (fighter_attack_profile_t){96, 18, 5, 1, 35, 2, 2, 10, 4};
92      case FIGHTER_ATTACK_DRAGON_PUNCH:
93        return (fighter_attack_profile_t){56, 20, 4, 4, 36, 2, 3, 10, 4};
94      case FIGHTER_ATTACK_JUMP_ATTACK:
95        return (fighter_attack_profile_t){52, 14, 2, 3, 4, 2, 1, 8, 3};
96      case FIGHTER_ATTACK_FORWARD_JUMP_ATTACK:
97        return (fighter_attack_profile_t){64, 16, 2, 3, 4, 2, 1, 8, 3};
98      case FIGHTER_ATTACK_BACK_JUMP_ATTACK:
99        return (fighter_attack_profile_t){52, 12, 2, 2, 4, 2, 1, 8, 3};
100     case FIGHTER_ATTACK_SWEEP:
101       return (fighter_attack_profile_t){58, 15, 4, 3, 18, 2, 2, 9, 4};
102     case FIGHTER_ATTACK_NONE:
103     default:
104       return (fighter_attack_profile_t){0, 0, 0, 0, 0, 0, 0, 0, 0};
105   }
106 }
107
108
109 static int fighter_attack_chip_damage(fighter_attack_profile_t profile) {
110   int damage = profile.damage / 3;
111
112   if (damage < 1) {
113     damage = 1;
114   }
115   return damage;
116 }
```

```
117
118
119 static void fighter_game_push_audio(fighter_audio_command_list_t *audio_commands,
120                                      fighter_audio_command_type_t type,
121                                      fighter_audio_track_t track) {
122   if (!audio_commands) {
123     return;
124   }
125
126   (void)fighter_audio_command_list_push(audio_commands, type, track);
127 }
128
129
130 static void fighter_player_sync_attack_visual_frames(
131     fighter_player_state_t *player) {
132   if (!player) {
133     return;
134   }
135
136   if (player->attack_phase == FIGHTER_ATTACK_PHASE_NONE) {
137     player->attack_visual_frames = 0;
138   } else {
139     player->attack_visual_frames =
140         player->attack_phase_frames > 0 ? player->attack_phase_frames : 1;
141   }
142 }
143
144
145 static void fighter_player_interrupt_attack(fighter_player_state_t *player) {
146   if (!player) {
147     return;
148   }
149
150   player->attack_phase = FIGHTER_ATTACK_PHASE_NONE;
151   player->attack_phase_frames = 0;
152   player->attack_has_connected = 0;
153   fighter_player_sync_attack_visual_frames(player);
154 }
155
156
157 static void fighter_player_enter_attack_phase(
158     fighter_player_state_t *player,
159     fighter_attack_phase_t phase,
160     int frames) {
161   if (!player) {
162     return;
163   }
164
165   player->attack_phase = phase;
166   player->attack_phase_frames = frames > 0 ? frames : 0;
167   fighter_player_sync_attack_visual_frames(player);
168 }
169
170
171 static fighter_attack_command_t fighter_normalize_attack_command(
172     fighter_attack_command_t command) {
173   if (command == FIGHTER_ATTACK_NONE) {
174     return FIGHTER_ATTACK_NORMAL;
175   }
176   return command;
177 }
178
179 static void fighter_player_begin_attack(fighter_player_state_t *player,
180                                         fighter_attack_command_t command) {
181   fighter_attack_profile_t profile;
182
183   if (!player) {
184     return;
185   }
186
187   command = fighter_normalize_attack_command(command);
188   profile = fighter_attack_profile(command);
189   player->last_attack = command;
```

```
  player->attack_has_connected = 0;
  player->combat_result = FIGHTER_COMBAT_RESULT_NONE;
  player->event_flags |= FIGHTER_PLAYER_EVENT_ATTACK_START;
  fighter_player_enter_attack_phase(player, FIGHTER_ATTACK_PHASE_STARTUP,
                                    profile.startup_frames);
}

/* Core attack state machine: startup -> active -> confirm/recovery. */
static void fighter_player_tick_attack_phase(fighter_player_state_t *player) {
  fighter_attack_profile_t profile;

  if (!player || player->attack_phase == FIGHTER_ATTACK_PHASE_NONE) {
    return;
  }

  if (player->attack_phase_frames > 0) {
    player->attack_phase_frames--;
  }

  if (player->attack_phase_frames > 0) {
    fighter_player_sync_attack_visual_frames(player);
    return;
  }

  profile = fighter_attack_profile(player->last_attack);
  switch (player->attack_phase) {
    case FIGHTER_ATTACK_PHASE_STARTUP:
      fighter_player_enter_attack_phase(player, FIGHTER_ATTACK_PHASE_ACTIVE,
                                        profile.active_frames);
      break;
    case FIGHTER_ATTACK_PHASE_ACTIVE:
      player->combat_result = FIGHTER_COMBAT_RESULT_WHIFF;
      fighter_player_enter_attack_phase(player, FIGHTER_ATTACK_PHASE_RECOVERY,
                                        profile.recovery_frames);
      break;
    case FIGHTER_ATTACK_PHASE_HIT_CONFIRM:
    case FIGHTER_ATTACK_PHASE_BLOCK_CONFIRM:
      fighter_player_enter_attack_phase(player, FIGHTER_ATTACK_PHASE_RECOVERY,
                                        profile.recovery_frames);
      break;
    case FIGHTER_ATTACK_PHASE_RECOVERY:
    case FIGHTER_ATTACK_PHASE_NONE:
    default:
      fighter_player_interrupt_attack(player);
      break;
  }
}


static void fighter_player_enter_hit(fighter_player_state_t *player,
                                     int hit_stun_frames,
                                     fighter_combat_result_t result) {
  if (!player) {
    return;
  }

  fighter_player_interrupt_attack(player);
  player->block_stun_frames = 0;
  player->hurt_visual_frames = hit_stun_frames;
  player->combat_result = result;
  player->event_flags |= FIGHTER_PLAYER_EVENT_HIT;
}

static void fighter_player_enter_block_stun(fighter_player_state_t *player,
                                            int block_stun_frames) {
  if (!player) {
    return;
  }

  player->hurt_visual_frames = 0;
  player->block_stun_frames = block_stun_frames;
  player->combat_result = FIGHTER_COMBAT_RESULT_BLOCKED;
  player->event_flags |= FIGHTER_PLAYER_EVENT_BLOCK;
```

```
}

static int fighter_player_can_enter_guard_state(
    const fighter_game_t *game,
    const fighter_player_state_t *player,
    const fighter_player_result_t *input) {
  if (!game || !player || !input) {
    return 0;
  }

  if (!input->guard_held) {
    return 0;
  }
  if (player->hp <= 0 || player->hurt_visual_frames > 0 ||
      player->block_stun_frames > 0 ||
      player->attack_phase != FIGHTER_ATTACK_PHASE_NONE) {
    return 0;
  }
  return !fighter_player_is_airborne(game, player);
}

static int fighter_player_can_crouch_guard(
    const fighter_game_t *game,
    const fighter_player_state_t *player,
    const fighter_player_result_t *input) {
  return fighter_player_can_enter_guard_state(game, player, input) &&
         input->crouch_held;
}

static int fighter_player_can_guard(const fighter_game_t *game,
                                    const fighter_player_state_t *player,
                                    const fighter_player_result_t *input) {
  return fighter_player_can_enter_guard_state(game, player, input);
}

static void fighter_game_spawn_fireball(fighter_game_t *game, int player_index) {
  fighter_projectile_state_t *projectile;
  fighter_player_state_t *player;
  int spawn_x;
  int spawn_y;

  if (!game || player_index < 0 || player_index >= FIGHTER_PLAYER_COUNT) {
    return;
  }

  projectile = &game->projectiles[player_index];
  if (projectile->active) {
    return;
  }

  player = &game->players[player_index];
  spawn_x = player->facing > 0
                ? player->x + game->config.player_width + 4
                : player->x - game->config.projectile_width - 4;
  spawn_y = player->y + game->config.player_height / 3;

  projectile->active = 1;
  projectile->owner_index = player_index;
  projectile->x = spawn_x;
  projectile->y = spawn_y;
  projectile->vx = player->facing * game->config.projectile_speed;
  projectile->character_id = player->character_id;
  projectile->anim_ticks = 0;
}

static void fighter_game_apply_projectile_hit(fighter_game_t *game,
                                              int attacker_index) {
  fighter_player_state_t *attacker;
  fighter_player_state_t *target;
  fighter_attack_profile_t profile;

  if (!game || attacker_index < 0 || attacker_index >= FIGHTER_PLAYER_COUNT) {
    return;
```

```
  }

  attacker = &game->players[attacker_index];
  target = &game->players[1 - attacker_index];
  profile = fighter_attack_profile(FIGHTER_ATTACK_FIREBALL);

  target->hp -= profile.damage;
  if (target->hp < 0) {
    target->hp = 0;
  }

  attacker->combat_result = FIGHTER_COMBAT_RESULT_HIT;
  attacker->event_flags |= FIGHTER_PLAYER_EVENT_HIT;

  if (target->hp == 0) {
    fighter_player_interrupt_attack(target);
    target->hurt_visual_frames = 0;
    target->block_stun_frames = 0;
    target->combat_result = FIGHTER_COMBAT_RESULT_HIT;
    target->event_flags |=FIGHTER_PLAYER_EVENT_HIT | FIGHTER_PLAYER_EVENT_KO;
  } else {
    fighter_player_enter_hit(target, profile.hit_stun_frames,
                             FIGHTER_COMBAT_RESULT_HIT);
  }
}

static void fighter_game_apply_projectile_block(fighter_game_t *game,
                                                int attacker_index) {
  fighter_player_state_t *attacker;
  fighter_player_state_t *target;
  fighter_attack_profile_t profile;
  int chip_damage;

  if (!game || attacker_index < 0 || attacker_index >= FIGHTER_PLAYER_COUNT) {
    return;
  }

  attacker = &game->players[attacker_index];
  target = &game->players[1 - attacker_index];
  profile = fighter_attack_profile(FIGHTER_ATTACK_FIREBALL);
  chip_damage = fighter_attack_chip_damage(profile);

  target->hp -= chip_damage;
  if (target->hp < 0) {
    target->hp = 0;
  }

  attacker->combat_result = FIGHTER_COMBAT_RESULT_BLOCKED;
  attacker->event_flags |= FIGHTER_PLAYER_EVENT_BLOCK;

  if (target->hp == 0) {
    fighter_player_interrupt_attack(target);
    target->hurt_visual_frames = 0;
    target->block_stun_frames = 0;
    target->combat_result = FIGHTER_COMBAT_RESULT_BLOCKED;
    target->event_flags |=
        FIGHTER_PLAYER_EVENT_BLOCK | FIGHTER_PLAYER_EVENT_KO;
  } else {
    fighter_player_enter_block_stun(target, profile.block_stun_frames);
  }
}

static void fighter_game_update_projectiles(
    fighter_game_t *game,
    const fighter_player_result_t inputs[FIGHTER_PLAYER_COUNT]) {
  int i;

  if (!game || !inputs) {
    return;
  }

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    fighter_projectile_state_t *projectile = &game->projectiles[i];
```

```
409
410     if (!projectile->active) {
411       continue;
412     }
413
414     projectile->x += projectile->vx;
415     projectile->anim_ticks++;
416
417     if (projectile->x + game->config.projectile_width < 0 ||
418         projectile->x >= game->config.screen_width) {
419       fighter_projectile_reset(projectile);
420     }
421   }
422
423   if (game->projectiles[0].active && game->projectiles[1].active &&
424       fighter_rects_overlap(game->projectiles[0].x, game->projectiles[0].y,
425                             game->config.projectile_width,
426                             game->config.projectile_height, game->projectiles[1].x,
427                             game->projectiles[1].y,
428                             game->config.projectile_width,
429                             game->config.projectile_height)) {
430     fighter_projectile_reset(&game->projectiles[0]);
431     fighter_projectile_reset(&game->projectiles[1]);
432     return;
433   }
434
435   for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
436     fighter_projectile_state_t *projectile = &game->projectiles[i];
437     fighter_player_state_t *target = &game->players[1 - i];
438     const fighter_player_result_t *target_input = &inputs[1 - i];
439
440     if (!projectile->active) {
441       continue;
442     }
443
444     if (!fighter_rects_overlap(projectile->x, projectile->y,
445                                game->config.projectile_width,
446                                game->config.projectile_height, target->x, target->y,
447                                game->config.player_width,
448                                game->config.player_height)) {
449       continue;
450     }
451
452     if (fighter_player_can_guard(game, target, target_input)) {
453       fighter_game_apply_projectile_block(game, i);
454     } else {
455       fighter_game_apply_projectile_hit(game, i);
456     }
457     fighter_projectile_reset(projectile);
458   }
459 }
460
461
462 static int fighter_player_controls_locked(const fighter_player_state_t *player) {
463   if (!player) {
464     return 1;
465   }
466
467   return player->hp <= 0 || player->hurt_visual_frames > 0 ||
468          player->block_stun_frames > 0 ||
469          player->attack_phase != FIGHTER_ATTACK_PHASE_NONE;
470 }
471
472 static void fighter_game_reset_round(fighter_game_t *game) {
473   int ground_y;
474   int i;
475
476   ground_y = fighter_player_ground_y(game);
477
478   memset(game->players, 0, sizeof(game->players));
479   for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
480     fighter_projectile_reset(&game->projectiles[i]);
481   }
```

```
482
483   game->players[0].x = game->config.screen_width / 4 - game->config.player_width / 2;
484   game->players[1].x =
485       (game->config.screen_width * 3) / 4 - game->config.player_width / 2;
486
487   game->players[0].y = ground_y;
488   game->players[1].y = ground_y;
489
490   game->players[0].vx = 0;
491   game->players[1].vx = 0;
492   game->players[0].vy = 0;
493   game->players[1].vy = 0;
494
495   game->players[0].hp = game->config.max_hp;
496   game->players[1].hp = game->config.max_hp;
497
498   game->players[0].facing = 1;
499   game->players[1].facing = -1;
500
501   game->players[0].last_attack = FIGHTER_ATTACK_NONE;
502   game->players[1].last_attack = FIGHTER_ATTACK_NONE;
503
504   game->players[0].attack_phase = FIGHTER_ATTACK_PHASE_NONE;
505   game->players[1].attack_phase = FIGHTER_ATTACK_PHASE_NONE;
506
507   game->players[0].visual_state = FIGHTER_VISUAL_STATE_IDLE;
508   game->players[1].visual_state = FIGHTER_VISUAL_STATE_IDLE;
509
510   game->players[0].character_id = FIGHTER_CHARACTER_RYU;
511   game->players[1].character_id = FIGHTER_CHARACTER_KEN;
512
513   game->players[0].state_frame = 0;
514   game->players[1].state_frame = 0;
515
516   game->round_timer_frames = (uint32_t)game->config.round_duration_frames;
517   game->winner = FIGHTER_WINNER_NONE;
518   game->finish_reason = FIGHTER_FINISH_REASON_NONE;
519 }
520
521 static void fighter_game_clear_frame_outputs(fighter_game_t *game) {
522   int i;
523
524   if (!game) {
525     return;
526   }
527
528   for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
529     game->players[i].combat_result = FIGHTER_COMBAT_RESULT_NONE;
530     game->players[i].event_flags = FIGHTER_PLAYER_EVENT_NONE;
531   }
532 }
533
534 static void fighter_game_enter_menu(fighter_game_t *game,
535                                     fighter_audio_command_list_t *audio_commands) {
536   fighter_game_reset_round(game);
537   game->state = FIGHTER_GAME_STATE_MENU;
538   game->state_frames = 0;
539   game->finish_reason = FIGHTER_FINISH_REASON_EXIT;
540   if (!game->menu_bgm_active) {
541     fighter_game_push_audio(audio_commands, FIGHTER_AUDIO_COMMAND_START_LOOP,
542                             FIGHTER_AUDIO_TRACK_MENU_BGM);
543     game->menu_bgm_active = 1;
544   }
545 }
546
547 static void fighter_game_start_round(fighter_game_t *game,
548                                      fighter_audio_command_list_t *audio_commands) {
549   fighter_game_reset_round(game);
550   game->state = FIGHTER_GAME_STATE_PLAYING;
551   game->state_frames = 0;
552   if (game->menu_bgm_active) {
553     fighter_game_push_audio(audio_commands, FIGHTER_AUDIO_COMMAND_STOP_LOOP,
554                             FIGHTER_AUDIO_TRACK_MENU_BGM);
```

```
    game->menu_bgm_active = 0;
  }
  fighter_game_push_audio(audio_commands, FIGHTER_AUDIO_COMMAND_PLAY_ONCE,
                          FIGHTER_AUDIO_TRACK_MENU_CONFIRM);
}

static void fighter_game_enter_game_over(fighter_game_t *game,
                                         fighter_winner_t winner,
                                         fighter_finish_reason_t reason,
                                         fighter_audio_command_list_t *audio_commands) {
  int i;

  if (!game) {
    return;
  }

  game->state = FIGHTER_GAME_STATE_GAME_OVER;
  game->state_frames = 0;
  game->winner = winner;
  game->finish_reason = reason;
  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    fighter_projectile_reset(&game->projectiles[i]);
  }

  if (winner == FIGHTER_WINNER_PLAYER1) {
    game->players[0].visual_state = FIGHTER_VISUAL_STATE_VICTORY;
    game->players[0].state_frame = 0;
    game->players[1].visual_state = FIGHTER_VISUAL_STATE_KO;
    game->players[1].state_frame = 0;
  } else if (winner == FIGHTER_WINNER_PLAYER2) {
    game->players[1].visual_state = FIGHTER_VISUAL_STATE_VICTORY;
    game->players[1].state_frame = 0;
    game->players[0].visual_state = FIGHTER_VISUAL_STATE_KO;
    game->players[0].state_frame = 0;
  } else if (winner == FIGHTER_WINNER_DRAW) {
    game->players[0].visual_state = FIGHTER_VISUAL_STATE_KO;
    game->players[1].visual_state = FIGHTER_VISUAL_STATE_KO;
    game->players[0].state_frame = 0;
    game->players[1].state_frame = 0;
  }

  fighter_game_push_audio(audio_commands, FIGHTER_AUDIO_COMMAND_PLAY_ONCE,FIGHTER_AUDIO_TRACK_GAME_OVER);
}

static void fighter_game_update_facing(fighter_game_t *game) {
  int p1_center;
  int p2_center;

  p1_center = fighter_player_center_x(game, &game->players[0]);
  p2_center = fighter_player_center_x(game, &game->players[1]);

  if (p1_center <= p2_center) {
    game->players[0].facing = 1;
    game->players[1].facing = -1;
  } else {
    game->players[0].facing = -1;
    game->players[1].facing = 1;
  }
}

static void fighter_game_resolve_overlap(fighter_game_t *game) {
  fighter_player_state_t *left_player;
  fighter_player_state_t *right_player;
  int overlap;
  int vertical_overlap;
  int push;
  int max_x;

  max_x = game->config.screen_width - game->config.player_width;
  if (game->players[0].x <= game->players[1].x) {
    left_player = &game->players[0];
    right_player = &game->players[1];
  } else {
```

```
    left_player = &game->players[1];
    right_player = &game->players[0];
  }

  overlap = left_player->x + game->config.player_width - right_player->x;
  if (overlap <= 0) {
    return;
  }
  vertical_overlap =
      fighter_player_vertical_overlap(game, left_player, right_player);
  if (vertical_overlap <= 0) {
    return;
  }

  push = overlap / 2 + 1;
  left_player->x = fighter_clamp_int(left_player->x - push, 0, max_x);
  right_player->x = fighter_clamp_int(right_player->x + push, 0, max_x);
}

static fighter_visual_state_t fighter_game_choose_visual_state(
    const fighter_game_t *game,
    const fighter_player_state_t *player,
    const fighter_player_result_t *input) {
  if (!game || !player) {
    return FIGHTER_VISUAL_STATE_IDLE;
  }

  if (player->visual_state == FIGHTER_VISUAL_STATE_VICTORY) {
    return FIGHTER_VISUAL_STATE_VICTORY;
  }
  if (player->hp <= 0) {
    return FIGHTER_VISUAL_STATE_KO;
  }
  if (player->hurt_visual_frames > 0) {
    return FIGHTER_VISUAL_STATE_HIT;
  }
  if (player->block_stun_frames > 0) {
    return FIGHTER_VISUAL_STATE_BLOCK_STUN;
  }
  if (player->attack_phase != FIGHTER_ATTACK_PHASE_NONE) {
    return FIGHTER_VISUAL_STATE_ATTACK;
  }
  if (fighter_player_is_airborne(game, player)) {
    return FIGHTER_VISUAL_STATE_JUMP;
  }
  if (input && fighter_player_can_crouch_guard(game, player, input)) {
    return FIGHTER_VISUAL_STATE_CROUCH_GUARD;
  }
  if (input && fighter_player_can_guard(game, player, input)) {
    return FIGHTER_VISUAL_STATE_GUARD;
  }
  if (input && input->crouch_held) {
    return FIGHTER_VISUAL_STATE_CROUCH;
  }
  if (input && (input->move_left ^ input->move_right)) {
    return FIGHTER_VISUAL_STATE_WALK;
  }
  return FIGHTER_VISUAL_STATE_IDLE;
}

static void fighter_game_update_visual_state(
    const fighter_game_t *game,
    fighter_player_state_t *player,
    const fighter_player_result_t *input) {
  fighter_visual_state_t next_state;

  if (!game || !player) {
    return;
  }

  next_state = fighter_game_choose_visual_state(game, player, input);
  if (player->visual_state != next_state) {
    player->visual_state = next_state;
```

```
    player->state_frame = 0;
  } else {
    player->state_frame++;
  }
}


static fighter_combat_result_t fighter_game_evaluate_contact(
    const fighter_game_t *game,
    int attacker_index,
    const fighter_player_result_t inputs[2]) {
  const fighter_player_state_t *attacker;
  const fighter_player_state_t *target;
  const fighter_player_result_t *target_input;
  fighter_attack_profile_t profile;
  int front_x;
  int target_center;
  int distance;

  if (!game || !inputs || attacker_index < 0 ||
      attacker_index >= FIGHTER_PLAYER_COUNT) {
    return FIGHTER_COMBAT_RESULT_NONE;
  }

  attacker = &game->players[attacker_index];
  target = &game->players[1 - attacker_index];
  target_input = &inputs[1 - attacker_index];
  if (attacker->hp <= 0 || target->hp <= 0 ||
      attacker->attack_phase != FIGHTER_ATTACK_PHASE_ACTIVE ||
      attacker->attack_has_connected) {
    return FIGHTER_COMBAT_RESULT_NONE;
  }

  profile = fighter_attack_profile(attacker->last_attack);
  if (attacker->last_attack == FIGHTER_ATTACK_FIREBALL || profile.damage == 0) {
    return FIGHTER_COMBAT_RESULT_NONE;
  }

  front_x = attacker->facing > 0 ? attacker->x + game->config.player_width : attacker->x;
  target_center = target->x + game->config.player_width / 2;
  distance = attacker->facing > 0 ? target_center - front_x : front_x - target_center;

  if (distance < -game->config.player_width / 2 || distance > profile.reach) {
    return FIGHTER_COMBAT_RESULT_NONE;
  }
  if (target->y - attacker->y > game->config.player_height / 2 ||
      attacker->y - target->y > game->config.player_height / 2) {
    return FIGHTER_COMBAT_RESULT_NONE;
  }
  if (fighter_player_can_guard(game, target, target_input)) {
    return FIGHTER_COMBAT_RESULT_BLOCKED;
  }
  return FIGHTER_COMBAT_RESULT_HIT;
}

static void fighter_game_apply_trade(fighter_game_t *game) {
  fighter_player_state_t *p1;
  fighter_player_state_t *p2;
  fighter_attack_profile_t p1_profile;
  fighter_attack_profile_t p2_profile;

  p1 = &game->players[0];
  p2 = &game->players[1];
  p1_profile = fighter_attack_profile(p1->last_attack);
  p2_profile = fighter_attack_profile(p2->last_attack);

  p1->hp -= p2_profile.damage;
  p2->hp -= p1_profile.damage;
  if (p1->hp < 0) {
    p1->hp = 0;
  }
  if (p2->hp < 0) {
    p2->hp = 0;
```

```
  }

  p1->attack_has_connected = 1;
  p2->attack_has_connected = 1;
  fighter_player_interrupt_attack(p1);
  fighter_player_interrupt_attack(p2);
  p1->combat_result = FIGHTER_COMBAT_RESULT_TRADE;
  p2->combat_result = FIGHTER_COMBAT_RESULT_TRADE;
  p1->event_flags |= FIGHTER_PLAYER_EVENT_HIT;
  p2->event_flags |= FIGHTER_PLAYER_EVENT_HIT;

  if (p1->hp > 0) {
    p1->hurt_visual_frames = p2_profile.hit_stun_frames;
  } else {
    p1->event_flags |= FIGHTER_PLAYER_EVENT_KO;
  }
  if (p2->hp > 0) {
    p2->hurt_visual_frames = p1_profile.hit_stun_frames;
  } else {
    p2->event_flags |= FIGHTER_PLAYER_EVENT_KO;
  }
}

static void fighter_game_apply_hit(fighter_game_t *game, int attacker_index) {
  fighter_player_state_t *attacker;
  fighter_player_state_t *target;
  fighter_attack_profile_t profile;

  attacker = &game->players[attacker_index];
  target = &game->players[1 - attacker_index];
  profile = fighter_attack_profile(attacker->last_attack);

  target->hp -= profile.damage;
  if (target->hp < 0) {
    target->hp = 0;
  }

  attacker->attack_has_connected = 1;
  fighter_player_enter_attack_phase(attacker, FIGHTER_ATTACK_PHASE_HIT_CONFIRM,profile.hit_confirm_frames)
    ;
  attacker->combat_result = FIGHTER_COMBAT_RESULT_HIT;
  attacker->event_flags |= FIGHTER_PLAYER_EVENT_HIT;

  if (target->hp == 0) {
    fighter_player_interrupt_attack(target);
    target->hurt_visual_frames = 0;
    target->block_stun_frames = 0;
    target->combat_result = FIGHTER_COMBAT_RESULT_HIT;
    target->event_flags |=
        FIGHTER_PLAYER_EVENT_HIT | FIGHTER_PLAYER_EVENT_KO;
  } else {
    fighter_player_enter_hit(target, profile.hit_stun_frames, FIGHTER_COMBAT_RESULT_HIT);
  }
}

static void fighter_game_apply_block(fighter_game_t *game, int attacker_index) {
  fighter_player_state_t *attacker;
  fighter_player_state_t *target;
  fighter_attack_profile_t profile;
  int chip_damage;

  attacker = &game->players[attacker_index];
  target = &game->players[1 - attacker_index];
  profile = fighter_attack_profile(attacker->last_attack);
  chip_damage = fighter_attack_chip_damage(profile);

  target->hp -= chip_damage;
  if (target->hp < 0) {
    target->hp = 0;
  }

  attacker->attack_has_connected = 1;
  fighter_player_enter_attack_phase(attacker,
```

```
                                    FIGHTER_ATTACK_PHASE_BLOCK_CONFIRM,
                                    profile.block_confirm_frames);
  attacker->combat_result = FIGHTER_COMBAT_RESULT_BLOCKED;
  attacker->event_flags |= FIGHTER_PLAYER_EVENT_BLOCK;

  if (target->hp == 0) {
    fighter_player_interrupt_attack(target);
    target->hurt_visual_frames = 0;
    target->block_stun_frames = 0;
    target->combat_result = FIGHTER_COMBAT_RESULT_BLOCKED;
    target->event_flags |=
        FIGHTER_PLAYER_EVENT_BLOCK | FIGHTER_PLAYER_EVENT_KO;
  } else {
    fighter_player_enter_block_stun(target, profile.block_stun_frames);
  }
}


static void fighter_game_resolve_attacks(
    fighter_game_t *game,
    const fighter_player_result_t inputs[2]) {
  fighter_combat_result_t p1_result;
  fighter_combat_result_t p2_result;

  if (!game || !inputs) {
    return;
  }

  p1_result = fighter_game_evaluate_contact(game, 0, inputs);
  p2_result = fighter_game_evaluate_contact(game, 1, inputs);

  if (p1_result == FIGHTER_COMBAT_RESULT_HIT &&
      p2_result == FIGHTER_COMBAT_RESULT_HIT) {
    fighter_game_apply_trade(game);
    return;
  }

  if (p1_result == FIGHTER_COMBAT_RESULT_HIT) {
    fighter_game_apply_hit(game, 0);
  } else if (p1_result == FIGHTER_COMBAT_RESULT_BLOCKED) {
    fighter_game_apply_block(game, 0);
  }

  if (game->players[1].hp <= 0) {
    return;
  }

  if (p2_result == FIGHTER_COMBAT_RESULT_HIT &&
      game->players[1].attack_phase == FIGHTER_ATTACK_PHASE_ACTIVE) {
    fighter_game_apply_hit(game, 1);
  } else if (p2_result == FIGHTER_COMBAT_RESULT_BLOCKED &&
             game->players[1].attack_phase == FIGHTER_ATTACK_PHASE_ACTIVE) {
    fighter_game_apply_block(game, 1);
  }
}


static void fighter_game_handle_round_end_from_hp(
    fighter_game_t *game,
    fighter_audio_command_list_t *audio_commands) {
  int p1_dead;
  int p2_dead;

  if (!game || game->state != FIGHTER_GAME_STATE_PLAYING) {
    return;
  }

  p1_dead = game->players[0].hp <= 0;
  p2_dead = game->players[1].hp <= 0;
  if (!p1_dead && !p2_dead) {
    return;
  }

```

```
  if (p1_dead) {
    game->players[0].event_flags |= FIGHTER_PLAYER_EVENT_KO;
  }
  if (p2_dead) {
    game->players[1].event_flags |= FIGHTER_PLAYER_EVENT_KO;
  }

  if (p1_dead && p2_dead) {
    fighter_game_enter_game_over(game, FIGHTER_WINNER_DRAW,
                                 FIGHTER_FINISH_REASON_DOUBLE_KO,
                                 audio_commands);
  } else if (p1_dead) {
    fighter_game_enter_game_over(game, FIGHTER_WINNER_PLAYER2,
                                 FIGHTER_FINISH_REASON_KO, audio_commands);
  } else {
    fighter_game_enter_game_over(game, FIGHTER_WINNER_PLAYER1,
                                 FIGHTER_FINISH_REASON_KO, audio_commands);
  }
}


static void fighter_game_handle_player(fighter_game_t *game,
                                       int player_index,
                                       const fighter_player_result_t inputs[2]) {
  fighter_player_state_t *player;
  const fighter_player_result_t *input;
  int ground_y;
  int max_x;
  int was_airborne;
  int is_airborne;
  int controls_locked;
  fighter_attack_phase_t previous_attack_phase;

  player = &game->players[player_index];
  input = &inputs[player_index];

  ground_y = fighter_player_ground_y(game);

  max_x = game->config.screen_width - game->config.player_width;
  was_airborne = fighter_player_is_airborne(game, player);

  if (!was_airborne) {
    player->vx = 0;
  }

  if (player->attack_cooldown_frames > 0) {
    player->attack_cooldown_frames--;
  }
  if (player->hurt_visual_frames > 0) {
    player->hurt_visual_frames--;
  }
  if (player->block_stun_frames > 0) {
    player->block_stun_frames--;
  }

  previous_attack_phase = player->attack_phase;
  fighter_player_tick_attack_phase(player);


  if (previous_attack_phase != FIGHTER_ATTACK_PHASE_ACTIVE &&
      player->attack_phase == FIGHTER_ATTACK_PHASE_ACTIVE) {
    if (player->last_attack == FIGHTER_ATTACK_DRAGON_PUNCH && player->vy >= 0) {
      player->vy = game->config.dragon_punch_lift_velocity;
    } else if (player->last_attack == FIGHTER_ATTACK_FIREBALL) {
      fighter_game_spawn_fireball(game, player_index);
    }
  }
  controls_locked = fighter_player_controls_locked(player);

  if (player->hp > 0 && !controls_locked) {
    if (input->jump_pressed && !was_airborne) {
      player->vy = game->config.jump_velocity;
      if (input->move_left && !input->move_right) {
```

```
        player->vx = -game->config.walk_speed;
      } else if (input->move_right && !input->move_left) {
        player->vx = game->config.walk_speed;
      } else {
        player->vx = 0;
      }
    }

    is_airborne = fighter_player_is_airborne(game, player);
    if (!is_airborne && !input->jump_held && !input->guard_held && !input->crouch_held) {
      if (input->move_left && !input->move_right) {
        player->vx = -game->config.walk_speed;
      } else if (input->move_right && !input->move_left) {
        player->vx = game->config.walk_speed;
      }
    }

    if (input->attack_pressed && player->attack_cooldown_frames == 0) {
      fighter_attack_command_t command = input->attack_command;
      int allow_attack = 0;

      if (was_airborne) {
        allow_attack = input->jump_held && command == FIGHTER_ATTACK_JUMP_ATTACK;
        if (allow_attack) {
          command = FIGHTER_ATTACK_JUMP_ATTACK;
        }
      } else if (!is_airborne) {
        allow_attack = command != FIGHTER_ATTACK_JUMP_ATTACK &&
                       command != FIGHTER_ATTACK_FORWARD_JUMP_ATTACK &&
                       command != FIGHTER_ATTACK_BACK_JUMP_ATTACK;
      }

      if (allow_attack) {
        player->attack_cooldown_frames = game->config.attack_cooldown_frames;
        fighter_player_begin_attack(player, command);
      }
    }
  }

  player->x += player->vx;
  player->x = fighter_clamp_int(player->x, 0, max_x);

  player->y += player->vy;
  if (player->y < ground_y) {
    player->vy += game->config.gravity;
  }
  if (player->y >= ground_y) {
    player->y = ground_y;
    if (was_airborne) {
      player->event_flags |= FIGHTER_PLAYER_EVENT_LAND;
    }
    player->vy = 0;
    player->vx = 0;
  }
}

static void fighter_game_tick_menu(fighter_game_t *game,
                                   const fighter_player_result_t inputs[2],
                                   fighter_audio_command_list_t *audio_commands) {
  int i;

  if (!game->menu_bgm_active) {
    fighter_game_push_audio(audio_commands, FIGHTER_AUDIO_COMMAND_START_LOOP,
                            FIGHTER_AUDIO_TRACK_MENU_BGM);
    game->menu_bgm_active = 1;
  }

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    if (inputs[i].any_input_pressed) {
      fighter_game_start_round(game, audio_commands);
      return;
    }
  }
```

```
}

/* Per-frame gameplay order matters: input, movement, collision, combat, then finish checks. */
static void fighter_game_tick_playing(fighter_game_t *game,
                                      const fighter_player_result_t inputs[2],
                                      fighter_audio_command_list_t *audio_commands) {
  int i;

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    if (inputs[i].exit_requested) {
      fighter_game_enter_menu(game, audio_commands);
      return;
    }
  }

  fighter_game_update_facing(game);
  fighter_game_handle_player(game, 0, inputs);
  fighter_game_handle_player(game, 1, inputs);
  fighter_game_resolve_overlap(game);
  fighter_game_update_facing(game);
  fighter_game_resolve_attacks(game, inputs);
  fighter_game_update_projectiles(game, inputs);

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    fighter_game_update_visual_state(game, &game->players[i], &inputs[i]);
  }

  fighter_game_handle_round_end_from_hp(game, audio_commands);
  if (game->state != FIGHTER_GAME_STATE_PLAYING) {
    return;
  }

  if (game->round_timer_frames > 0) {
    game->round_timer_frames--;
  }

  if (game->round_timer_frames == 0) {
    fighter_winner_t winner = FIGHTER_WINNER_DRAW;

    if (game->players[0].hp > game->players[1].hp) {
      winner = FIGHTER_WINNER_PLAYER1;
    } else if (game->players[1].hp > game->players[0].hp) {
      winner = FIGHTER_WINNER_PLAYER2;
    }
    fighter_game_enter_game_over(game, winner, FIGHTER_FINISH_REASON_TIME_OUT, audio_commands);
  }
}

static void fighter_game_tick_game_over(fighter_game_t *game,
                                        const fighter_player_result_t inputs[2],
                                        fighter_audio_command_list_t *audio_commands) {
  int i;

  if (!fighter_game_game_over_ready(game)) {
    return;
  }

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    if (inputs[i].exit_requested) {
      fighter_game_enter_menu(game, audio_commands);
      return;
    }
    if (inputs[i].attack_pressed || inputs[i].guard_pressed) {
      fighter_game_start_round(game, audio_commands);
      return;
    }
  }
}

void fighter_game_config_default(fighter_game_config_t *config) {
  if (!config) {
    return;
  }
```

```
1138
1139    memset(config, 0, sizeof(*config));
1140    config->screen_width = 640;
1141    config->screen_height = 480;
1142    config->floor_y = 400;
1143    config->player_width = 48;
1144    config->player_height = 96;
1145    config->projectile_width = 28;
1146    config->projectile_height = 20;
1147    config->projectile_speed = 6;
1148    config->walk_speed = 3;
1149    config->jump_velocity = -14;
1150    config->gravity = 1;
1151    config->max_hp = 100;
1152    config->round_duration_frames = 99 * 60;
1153    config->menu_anim_period_frames = 20;
1154    config->game_over_anim_frames = 120;
1155    config->attack_cooldown_frames = 14;
1156    config->dragon_punch_lift_velocity = -9;
1157    config->attack_visual_frames = 6;
1158    config->hurt_visual_frames = 8;
1159  }
1160
1161  void fighter_game_init(fighter_game_t *game, const fighter_game_config_t *config) {
1162    if (!game) {
1163      return;
1164    }
1165
1166    memset(game, 0, sizeof(*game));
1167    if (config) {
1168      game->config = *config;
1169    } else {
1170      fighter_game_config_default(&game->config);
1171    }
1172
1173    fighter_game_reset_round(game);
1174    game->state = FIGHTER_GAME_STATE_MENU;
1175  }
1176
1177  void fighter_game_tick(fighter_game_t *game,
1178                         const fighter_player_result_t inputs[FIGHTER_PLAYER_COUNT],
1179                         fighter_audio_command_list_t *audio_commands) {
1180    fighter_game_state_t previous_state;
1181
1182    if (!game || !inputs) {
1183      return;
1184    }
1185
1186    if (audio_commands) {
1187      fighter_audio_command_list_clear(audio_commands);
1188    }
1189
1190    fighter_game_clear_frame_outputs(game);
1191    previous_state = game->state;
1192    game->frame_counter++;
1193
1194    switch (game->state) {
1195      case FIGHTER_GAME_STATE_MENU:
1196        fighter_game_tick_menu(game, inputs, audio_commands);
1197        break;
1198      case FIGHTER_GAME_STATE_PLAYING:
1199        fighter_game_tick_playing(game, inputs, audio_commands);
1200        break;
1201      case FIGHTER_GAME_STATE_GAME_OVER:
1202        fighter_game_tick_game_over(game, inputs, audio_commands);
1203        break;
1204      default:
1205        break;
1206    }
1207
1208    if (game->state == previous_state) {
1209      game->state_frames++;
1210    }
```

```
}

int fighter_game_menu_animation_frame(const fighter_game_t *game) {
  if (!game || game->config.menu_anim_period_frames <= 0) {
    return 0;
  }

  return (int)((game->frame_counter /(uint32_t)game->config.menu_anim_period_frames) & 1U);

}

int fighter_game_game_over_ready(const fighter_game_t *game) {
  if (!game || game->state != FIGHTER_GAME_STATE_GAME_OVER) {
    return 0;
  }

  return game->state_frames >= (uint32_t)game->config.game_over_anim_frames;
}

int fighter_game_round_seconds_remaining(const fighter_game_t *game) {
  if (!game) {
    return 0;
  }

  return (int)((game->round_timer_frames + 59U) / 60U);
}
```

A.1.3 sw/input/fighter_gamepad.c

```
#include "fighter_gamepad.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <linux/input.h>


static int button_pressed(bool current, bool previous) {
  return current && !previous;
}


static fighter_attack_command_t attack_command_from_gamepad(
    const fighter_gamepad_buttons_t *buttons) {


  if (!buttons) {
    return FIGHTER_ATTACK_NORMAL;
  }

  if (buttons->fireball) {
    return FIGHTER_ATTACK_FIREBALL;
  }

  if (buttons->dragon_punch) {
    return FIGHTER_ATTACK_DRAGON_PUNCH;
  }

  if (buttons->up) {
    return FIGHTER_ATTACK_JUMP_ATTACK;
  }

  if (buttons->down) {
    return FIGHTER_ATTACK_SWEEP;
  }

  if (buttons->left) {
    return FIGHTER_ATTACK_FIREBALL;
  }
```

```
42
43    if (buttons->right) {
44      return FIGHTER_ATTACK_DRAGON_PUNCH;
45    }
46
47
48    return FIGHTER_ATTACK_NORMAL;
49  }
50
51
52  void fighter_gamepad_init(fighter_gamepad_t *gamepad) {
53
54    if (!gamepad) {
55      return;
56    }
57
58
59    memset(gamepad, 0, sizeof(*gamepad));
60    gamepad->fd = -1;
61  }
62
63
64  int fighter_gamepad_open(fighter_gamepad_t *gamepad, const char *device_path) {
65
66    if (!gamepad || !device_path || device_path[0] == '\0') {
67      return -1;
68    }
69
70
71    fighter_gamepad_close(gamepad);
72
73
74    gamepad->fd = open(device_path, O_RDONLY | O_NONBLOCK);
75
76    if (gamepad->fd < 0) {
77      return -1;
78    }
79
80    gamepad->connected = 1;
81
82
83    snprintf(gamepad->device_path, sizeof(gamepad->device_path), "%s", device_path);
84
85
86    memset(&gamepad->current_buttons, 0, sizeof(gamepad->current_buttons));
87    memset(&gamepad->previous_buttons, 0, sizeof(gamepad->previous_buttons));
88
89    return 0;
90  }
91
92
93  void fighter_gamepad_close(fighter_gamepad_t *gamepad) {
94
95    if (!gamepad) {
96      return;
97    }
98
99
100   if (gamepad->fd >= 0) {
101     close(gamepad->fd);
102   }
103
104   gamepad->fd = -1;
105
106   gamepad->connected = 0;
107
108   gamepad->device_path[0] = '\0';
109
110
111   memset(&gamepad->current_buttons, 0, sizeof(gamepad->current_buttons));
112   memset(&gamepad->previous_buttons, 0, sizeof(gamepad->previous_buttons));
113 }
114
```

```
115
116 static void fighter_gamepad_handle_abs(fighter_gamepad_t *gamepad, unsigned short code,int value) {
117
118   const int low_threshold = 64;
119
120   const int high_threshold = 190;
121
122
123   if (!gamepad) {
124     return;
125   }
126
127
128   if (code == ABS_X) {
129
130     gamepad->current_buttons.left = value < low_threshold;
131
132     gamepad->current_buttons.right = value > high_threshold;
133   }
134
135   else if (code == ABS_Y) {
136
137     gamepad->current_buttons.up = value < low_threshold;
138
139     gamepad->current_buttons.down = value > high_threshold;
140   }
141 }
142
143
144 static void fighter_gamepad_handle_key(fighter_gamepad_t *gamepad,unsigned short code,int value) {
145   bool pressed;
146
147   if (!gamepad) {
148     return;
149   }
150
151
152   pressed = value != 0;
153
154   switch (code) {
155
156     case BTN_THUMB:
157       gamepad->current_buttons.attack = pressed;
158       break;
159
160     case BTN_THUMB2:
161       gamepad->current_buttons.guard = pressed;
162       break;
163
164     case BTN_TRIGGER:
165       gamepad->current_buttons.fireball = pressed;
166       break;
167
168     case BTN_TOP:
169       gamepad->current_buttons.dragon_punch = pressed;
170       break;
171
172     case BTN_BASE3:
173       gamepad->current_buttons.exit_game = pressed;
174       break;
175
176     case BTN_BASE4:
177       gamepad->current_buttons.start = pressed;
178       break;
179
180     case BTN_TOP2:
181       gamepad->current_buttons.guard = pressed;
182       break;
183
184     case BTN_PINKIE:
185       gamepad->current_buttons.attack = pressed;
186       break;
187
```

```
    default:
      break;
  }
}


/* Nonblocking event drain keeps the frame loop responsive while preserving edge detection. */
static int fighter_gamepad_drain_events(fighter_gamepad_t *gamepad) {

  if (!gamepad || gamepad->fd < 0) {
    return -1;
  }


  while (1) {

    struct input_event event;

    ssize_t bytes_read = read(gamepad->fd, &event, sizeof(event));


    if (bytes_read < 0) {

      if (errno == EAGAIN || errno == EWOULDBLOCK) {
        return 0;
      }

      if (errno == EINTR) {
        continue;
      }

      return -1;
    }

    if (bytes_read != (ssize_t)sizeof(event)) {
      return -1;
    }


    if (event.type == EV_ABS) {
      fighter_gamepad_handle_abs(gamepad, event.code, event.value);

    } else if (event.type == EV_KEY) {
      fighter_gamepad_handle_key(gamepad, event.code, event.value);
    }
  }
}


int fighter_gamepad_update(fighter_gamepad_t *gamepad,fighter_player_result_t *result) {

  fighter_gamepad_buttons_t *current;

  fighter_gamepad_buttons_t *previous;

  int attack_edge;
  int fireball_edge;
  int dragon_punch_edge;
  int start_edge;
  int guard_edge;


  if (!gamepad || !result || !gamepad->connected) {
    return -1;
  }


  memset(result, 0, sizeof(*result));


  if (fighter_gamepad_drain_events(gamepad) != 0) {
    return -1;
  }
```

```
261
262
263     current = &gamepad->current_buttons;
264
265     previous = &gamepad->previous_buttons;
266
267     attack_edge = button_pressed(current->attack, previous->attack);
268
269     fireball_edge = button_pressed(current->fireball, previous->fireball);
270
271     dragon_punch_edge = button_pressed(current->dragon_punch, previous->dragon_punch);
272
273     start_edge = button_pressed(current->start, previous->start);
274
275     guard_edge = button_pressed(current->guard, previous->guard);
276
277
278     result->move_left = current->left && !current->right;
279     result->move_right = current->right && !current->left;
280
281
282     result->move_left_pressed =button_pressed(result->move_left, previous->left);
283     result->move_right_pressed =button_pressed(result->move_right, previous->right);
284
285
286     result->jump_held = current->up;
287
288     result->jump_pressed = button_pressed(current->up, previous->up);
289
290     result->crouch_held = current->down;
291     result->crouch_pressed = button_pressed(current->down, previous->down);
292
293     result->guard_held = current->guard;
294     result->guard_pressed = guard_edge || start_edge;
295
296
297     result->exit_requested = button_pressed(current->exit_game, previous->exit_game);
298
299
300     result->attack_pressed = attack_edge || fireball_edge || dragon_punch_edge;
301
302
303     result->any_input_active =
304         current->up || current->down || current->left || current->right ||
305         current->attack || current->guard || current->fireball ||
306         current->dragon_punch || current->start || current->exit_game;
307
308
309     result->any_input_pressed =
310         button_pressed(current->up, previous->up) ||
311         button_pressed(current->down, previous->down) ||
312         button_pressed(current->left, previous->left) ||
313         button_pressed(current->right, previous->right) ||
314         attack_edge ||
315         guard_edge ||
316         fireball_edge || dragon_punch_edge ||
317         start_edge ||
318         button_pressed(current->exit_game, previous->exit_game);
319
320
321     if (dragon_punch_edge) {
322       result->attack_command = FIGHTER_ATTACK_DRAGON_PUNCH;
323
324     } else if (fireball_edge) {
325       result->attack_command = FIGHTER_ATTACK_FIREBALL;
326
327     } else if (attack_edge) {
328       result->attack_command = attack_command_from_gamepad(current);
329
330     } else {
331       result->attack_command = FIGHTER_ATTACK_NONE;
332     }
333
```

```
334
335   gamepad->previous_buttons = gamepad->current_buttons;
336   return 0;
337 }
```

A.1.4 sw/input/fighter_input.c

```
1  #include "fighter_input.h"
2
3  #include <string.h>
4
5  static fighter_button_state_t fighter_buttons_from_report(
6      const usb_hid_keyboard_report_t *report) {
7    fighter_button_state_t buttons;
8
9    memset(&buttons, 0, sizeof(buttons));
10   if (!report) {
11     return buttons;
12   }
13
14   buttons.up = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_W);
15   buttons.down = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_S);
16   buttons.left = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_A);
17   buttons.right = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_D);
18   buttons.attack = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_J);
19   buttons.guard = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_K);
20   buttons.exit_game = usb_hid_keyboard_report_contains(report, FIGHTER_HID_KEY_L);
21
22   return buttons;
23 }
24
25 static int fighter_button_pressed(bool current, bool previous) {
26   return current && !previous;
27 }
28
29 void fighter_menu_parser_init(fighter_menu_parser_t *parser) {
30   if (!parser) {
31     return;
32   }
33
34   memset(parser, 0, sizeof(*parser));
35   parser->selected_item = FIGHTER_MENU_ITEM_START;
36 }
37
38 void fighter_player_parser_init(fighter_player_parser_t *parser) {
39   if (!parser) {
40     return;
41   }
42
43   memset(parser, 0, sizeof(*parser));
44 }
45
46 void fighter_menu_parser_update(fighter_menu_parser_t *parser,
47                                 const usb_hid_keyboard_report_t *report,
48                                 fighter_menu_result_t *result) {
49   fighter_button_state_t buttons;
50
51   if (!parser || !result) {
52     return;
53   }
54
55   buttons = fighter_buttons_from_report(report);
56
57   result->action = FIGHTER_MENU_ACTION_NONE;
58   result->selected_item = parser->selected_item;
59
60   if (fighter_button_pressed(buttons.left, parser->previous_buttons.left)) {
61     parser->selected_item =
62         parser->selected_item == FIGHTER_MENU_ITEM_START ? FIGHTER_MENU_ITEM_EXIT
63                                                          : FIGHTER_MENU_ITEM_START;
```

```
    result->action = FIGHTER_MENU_ACTION_MOVE_LEFT;
  } else if (fighter_button_pressed(buttons.right, parser->previous_buttons.right)) {
    parser->selected_item =
        parser->selected_item == FIGHTER_MENU_ITEM_START ? FIGHTER_MENU_ITEM_EXIT
                                                         : FIGHTER_MENU_ITEM_START;
    result->action = FIGHTER_MENU_ACTION_MOVE_RIGHT;
  } else if (fighter_button_pressed(buttons.attack, parser->previous_buttons.attack)) {
    result->action = FIGHTER_MENU_ACTION_CONFIRM;
  }

  result->selected_item = parser->selected_item;
  parser->previous_buttons = buttons;
}

void fighter_player_parser_update(fighter_player_parser_t *parser,
                                  const usb_hid_keyboard_report_t *report,
                                  fighter_player_result_t *result) {
  fighter_button_state_t buttons;
  int attack_edge;

  if (!parser || !result) {
    return;
  }

  buttons = fighter_buttons_from_report(report);
  attack_edge = fighter_button_pressed(buttons.attack, parser->previous_buttons.attack);

  memset(result, 0, sizeof(*result));

  result->move_left = buttons.left && !buttons.right;
  result->move_right = buttons.right && !buttons.left;
  result->move_left_pressed =
      fighter_button_pressed(result->move_left, parser->previous_buttons.left);
  result->move_right_pressed =
      fighter_button_pressed(result->move_right, parser->previous_buttons.right);
  result->jump_held = buttons.up;
  result->jump_pressed = fighter_button_pressed(buttons.up, parser->previous_buttons.up);
  result->crouch_held = buttons.down;
  result->crouch_pressed =
      fighter_button_pressed(buttons.down, parser->previous_buttons.down);
  result->guard_held = buttons.guard;
  result->guard_pressed =
      fighter_button_pressed(buttons.guard, parser->previous_buttons.guard);
  result->exit_requested =
      fighter_button_pressed(buttons.exit_game, parser->previous_buttons.exit_game);
  result->attack_pressed = attack_edge;
  result->any_input_active = buttons.up || buttons.down || buttons.left || buttons.right ||
                             buttons.attack || buttons.guard || buttons.exit_game;
  result->any_input_pressed =
      fighter_button_pressed(result->any_input_active,
                             parser->previous_buttons.up || parser->previous_buttons.down ||
                                 parser->previous_buttons.left ||
                                 parser->previous_buttons.right ||
                                 parser->previous_buttons.attack ||
                                 parser->previous_buttons.guard ||
                                 parser->previous_buttons.exit_game);
  result->attack_command = FIGHTER_ATTACK_NONE;

  if (attack_edge) {
    if (buttons.up && buttons.right) {
      result->attack_command = FIGHTER_ATTACK_FORWARD_JUMP_ATTACK;
    } else if (buttons.up && buttons.left) {
      result->attack_command = FIGHTER_ATTACK_BACK_JUMP_ATTACK;
    } else if (buttons.up) {
      result->attack_command = FIGHTER_ATTACK_JUMP_ATTACK;
    } else if (buttons.down) {
      result->attack_command = FIGHTER_ATTACK_SWEEP;
    } else if (buttons.left) {
      result->attack_command = FIGHTER_ATTACK_FIREBALL;
    } else if (buttons.right) {
      result->attack_command = FIGHTER_ATTACK_DRAGON_PUNCH;
    } else {
      result->attack_command = FIGHTER_ATTACK_NORMAL;
```

```
    }
  }

  parser->previous_buttons = buttons;
}

const char *fighter_attack_command_name(fighter_attack_command_t command) {
  switch (command) {
    case FIGHTER_ATTACK_NONE:
      return "none";
    case FIGHTER_ATTACK_NORMAL:
      return "normal_attack";
    case FIGHTER_ATTACK_FIREBALL:
      return "fireball";
    case FIGHTER_ATTACK_DRAGON_PUNCH:
      return "dragon_punch";
    case FIGHTER_ATTACK_JUMP_ATTACK:
      return "jump_attack";
    case FIGHTER_ATTACK_FORWARD_JUMP_ATTACK:
      return "forward_jump_attack";
    case FIGHTER_ATTACK_BACK_JUMP_ATTACK:
      return "back_jump_attack";
    case FIGHTER_ATTACK_SWEEP:
      return "sweep";
    default:
      return "unknown";
  }
}

const char *fighter_menu_item_name(fighter_menu_item_t item) {
  switch (item) {
    case FIGHTER_MENU_ITEM_START:
      return "start_game";
    case FIGHTER_MENU_ITEM_EXIT:
      return "exit_game";
    default:
      return "unknown";
  }
}
```

A.1.5 sw/input/hid_keyboard_report.c

```
#include "hid_keyboard_report.h"

#include <string.h>

void usb_hid_keyboard_report_clear(usb_hid_keyboard_report_t *report) {
  if (!report) {
    return;
  }

  memset(report, 0, sizeof(*report));
}

int usb_hid_keyboard_report_add_key(usb_hid_keyboard_report_t *report,
                                    uint8_t keycode) {
  int i;

  if (!report || keycode == 0) {
    return -1;
  }

  for (i = 0; i < 6; ++i) {
    if (report->keycode[i] == keycode) {
      return 0;
    }
  }

  for (i = 0; i < 6; ++i) {
    if (report->keycode[i] == 0) {
```

```
      report->keycode[i] = keycode;
      return 0;
    }
  }

  return -1;
}

int usb_hid_keyboard_report_contains(const usb_hid_keyboard_report_t *report,
                                     uint8_t keycode) {
  int i;

  if (!report || keycode == 0) {
    return 0;
  }

  for (i = 0; i < 6; ++i) {
    if (report->keycode[i] == keycode) {
      return 1;
    }
  }

  return 0;
}

int usb_hid_keyboard_report_is_empty(const usb_hid_keyboard_report_t *report) {
  int i;

  if (!report) {
    return 1;
  }

  if (report->modifiers != 0) {
    return 0;
  }

  for (i = 0; i < 6; ++i) {
    if (report->keycode[i] != 0) {
      return 0;
    }
  }

  return 1;
}
```

A.1.6 sw/input/usb_hid_keyboard.c

```
#include "usb_hid_keyboard.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define USB_CLASS_HID 0x03
#define USB_SUBCLASS_BOOT 0x01
#define USB_PROTOCOL_KEYBOARD 0x01

static void usb_hid_keyboard_device_close(usb_hid_keyboard_device_t *device) {
  if (!device || !device->connected) {
    return;
  }

  if (device->handle) {
    libusb_release_interface(device->handle, device->interface_number);
    libusb_close(device->handle);
  }

  memset(device, 0, sizeof(*device));
}

```

```
24 static int usb_hid_keyboard_find_endpoint(const struct libusb_interface_descriptor *desc,
25                                            uint8_t *endpoint_address) {
26   int i;
27
28   for (i = 0; i < desc->bNumEndpoints; ++i) {
29     const struct libusb_endpoint_descriptor *endpoint = &desc->endpoint[i];
30     const uint8_t endpoint_type = endpoint->bmAttributes & LIBUSB_TRANSFER_TYPE_MASK;
31     const int is_interrupt_in = endpoint_type == LIBUSB_TRANSFER_TYPE_INTERRUPT &&
32                                 (endpoint->bEndpointAddress & LIBUSB_ENDPOINT_DIR_MASK) ==
33                                     LIBUSB_ENDPOINT_IN;
34
35     if (is_interrupt_in) {
36       *endpoint_address = endpoint->bEndpointAddress;
37       return 0;
38     }
39   }
40
41   return -1;
42 }
43
44 static int usb_hid_keyboard_probe_device(libusb_device *usb_device,
45                                          usb_hid_keyboard_device_t *device) {
46   struct libusb_device_descriptor device_desc;
47   struct libusb_config_descriptor *config = NULL;
48   libusb_device_handle *handle = NULL;
49   uint8_t endpoint_address = 0;
50   int interface_number = -1;
51   int rc;
52   int interface_index;
53
54   memset(device, 0, sizeof(*device));
55
56   rc = libusb_get_device_descriptor(usb_device, &device_desc);
57   if (rc != 0) {
58     return rc;
59   }
60
61   rc = libusb_get_active_config_descriptor(usb_device, &config);
62   if (rc != 0) {
63     return rc;
64   }
65
66   for (interface_index = 0; interface_index < config->bNumInterfaces; ++interface_index) {
67     const struct libusb_interface *interface = &config->interface[interface_index];
68
69     if (interface->num_altsetting < 1) {
70       continue;
71     }
72
73     const struct libusb_interface_descriptor *desc = &interface->altsetting[0];
74     if (desc->bInterfaceClass != USB_CLASS_HID ||
75         desc->bInterfaceSubClass != USB_SUBCLASS_BOOT ||
76         desc->bInterfaceProtocol != USB_PROTOCOL_KEYBOARD) {
77       continue;
78     }
79
80     if (usb_hid_keyboard_find_endpoint(desc, &endpoint_address) != 0) {
81       continue;
82     }
83
84     interface_number = desc->bInterfaceNumber;
85     break;
86   }
87
88   if (interface_number < 0) {
89     libusb_free_config_descriptor(config);
90     return -1;
91   }
92
93   rc = libusb_open(usb_device, &handle);
94   if (rc != 0) {
95     libusb_free_config_descriptor(config);
96     return rc;
```

```
  }

  libusb_set_auto_detach_kernel_driver(handle, 1);

  rc = libusb_claim_interface(handle, interface_number);
  if (rc != 0) {
    libusb_close(handle);
    libusb_free_config_descriptor(config);
    return rc;
  }

  device->handle = handle;
  device->endpoint_address = endpoint_address;
  device->interface_number = interface_number;
  device->bus_number = libusb_get_bus_number(usb_device);
  device->device_address = libusb_get_device_address(usb_device);
  device->connected = 1;

  if (device_desc.iProduct != 0) {
    int len = libusb_get_string_descriptor_ascii(handle, device_desc.iProduct,
                                                 (unsigned char *)device->product_name,
                                                 (int)sizeof(device->product_name) - 1);
    if (len > 0) {
      device->product_name[len] = '\0';
    }
  }

  if (device->product_name[0] == '\0') {
    snprintf(device->product_name, sizeof(device->product_name),
             "bus%d-dev%d", device->bus_number, device->device_address);
  }

  libusb_free_config_descriptor(config);
  return 0;
}

int usb_hid_keyboard_manager_init(usb_hid_keyboard_manager_t *manager,
                                  size_t max_devices) {
  libusb_device **device_list = NULL;
  ssize_t device_count;
  ssize_t i;
  size_t limit;
  int rc;

  if (!manager) {
    return LIBUSB_ERROR_INVALID_PARAM;
  }

  memset(manager, 0, sizeof(*manager));

  rc = libusb_init(&manager->context);
  if (rc != 0) {
    return rc;
  }

  device_count = libusb_get_device_list(manager->context, &device_list);
  if (device_count < 0) {
    libusb_exit(manager->context);
    memset(manager, 0, sizeof(*manager));
    return (int)device_count;
  }

  limit = max_devices;
  if (limit > USB_HID_KEYBOARD_MAX_DEVICES) {
    limit = USB_HID_KEYBOARD_MAX_DEVICES;
  }

  for (i = 0; i < device_count && manager->device_count < limit; ++i) {
    usb_hid_keyboard_device_t candidate;
    if (usb_hid_keyboard_probe_device(device_list[i], &candidate) == 0) {
      manager->devices[manager->device_count++] = candidate;
    }
  }
```

```
170
171   libusb_free_device_list(device_list, 1);
172
173   if (manager->device_count == 0) {
174     usb_hid_keyboard_manager_close(manager);
175     return -1;
176   }
177
178   return 0;
179 }
180
181 void usb_hid_keyboard_manager_close(usb_hid_keyboard_manager_t *manager) {
182   size_t i;
183
184   if (!manager) {
185     return;
186   }
187
188   for (i = 0; i < manager->device_count; ++i) {
189     usb_hid_keyboard_device_close(&manager->devices[i]);
190   }
191
192   if (manager->context) {
193     libusb_exit(manager->context);
194   }
195
196   memset(manager, 0, sizeof(*manager));
197 }
198
199 int usb_hid_keyboard_manager_poll(usb_hid_keyboard_manager_t *manager,
200                                   usb_hid_keyboard_report_t *reports,
201                                   size_t report_capacity,
202                                   int timeout_ms) {
203   size_t i;
204
205   if (!manager || !reports) {
206     return LIBUSB_ERROR_INVALID_PARAM;
207   }
208
209   if (report_capacity < manager->device_count) {
210     return LIBUSB_ERROR_INVALID_PARAM;
211   }
212
213   for (i = 0; i < manager->device_count; ++i) {
214     usb_hid_keyboard_device_t *device = &manager->devices[i];
215     int transferred = 0;
216     int rc;
217
218     if (!device->connected) {
219       memset(&reports[i], 0, sizeof(reports[i]));
220       continue;
221     }
222
223     rc = libusb_interrupt_transfer(device->handle, device->endpoint_address,
224                                    (unsigned char *)&device->last_report,
225                                    (int)sizeof(device->last_report), &transferred,
226                                    timeout_ms);
227
228     if (rc == LIBUSB_ERROR_TIMEOUT) {
229       reports[i] = device->last_report;
230       continue;
231     }
232
233     if (rc == LIBUSB_ERROR_NO_DEVICE) {
234       fprintf(stderr, "keyboard disconnected: %s\n", device->product_name);
235       usb_hid_keyboard_device_close(device);
236       memset(&reports[i], 0, sizeof(reports[i]));
237       continue;
238     }
239
240     if (rc != 0) {
241       fprintf(stderr, "keyboard poll error on %s: %s\n", device->product_name,
242               libusb_error_name(rc));
```

```
      reports[i] = device->last_report;
      continue;
    }

    if (transferred != (int)sizeof(device->last_report)) {
      reports[i] = device->last_report;
      continue;
    }

    reports[i] = device->last_report;
  }

  return (int)manager->device_count;
}
```

A.1.7 sw/render_if/fighter_mmio.c

```
#include "fighter_mmio.h"

#include <string.h>

void fighter_mmio_encode(const fighter_game_t *game,
                         uint32_t regs[FIGHTER_MMIO_REG_COUNT]) {
  uint32_t game_state_word;

  if (!game || !regs) {
    return;
  }

  memset(regs, 0, sizeof(uint32_t) * FIGHTER_MMIO_REG_COUNT);

  game_state_word = (uint32_t)game->state & 0x3U;
  game_state_word |=
      (uint32_t)(fighter_game_menu_animation_frame(game) & 0x1) << 8;
  game_state_word |=
      (uint32_t)(fighter_game_game_over_ready(game) & 0x1) << 9;

  regs[FIGHTER_MMIO_REG_GAME_STATE] = game_state_word;
  regs[FIGHTER_MMIO_REG_PLAYER1_X] = (uint32_t)game->players[0].x;
  regs[FIGHTER_MMIO_REG_PLAYER1_Y] = (uint32_t)game->players[0].y;
  regs[FIGHTER_MMIO_REG_PLAYER1_STATE] =
      (uint32_t)game->players[0].visual_state;
  regs[FIGHTER_MMIO_REG_PLAYER2_X] = (uint32_t)game->players[1].x;
  regs[FIGHTER_MMIO_REG_PLAYER2_Y] = (uint32_t)game->players[1].y;
  regs[FIGHTER_MMIO_REG_PLAYER2_STATE] =
      (uint32_t)game->players[1].visual_state;
  regs[FIGHTER_MMIO_REG_PLAYER1_HP] = (uint32_t)game->players[0].hp;
  regs[FIGHTER_MMIO_REG_PLAYER2_HP] = (uint32_t)game->players[1].hp;
  regs[FIGHTER_MMIO_REG_ROUND_TIMER] =
      (uint32_t)fighter_game_round_seconds_remaining(game);
  regs[FIGHTER_MMIO_REG_WINNER] = (uint32_t)game->winner;
  regs[FIGHTER_MMIO_REG_PLAYER1_FACING] =
      game->players[0].facing > 0 ? 1U : 0U;
  regs[FIGHTER_MMIO_REG_PLAYER2_FACING] =
      game->players[1].facing > 0 ? 1U : 0U;
  regs[FIGHTER_MMIO_REG_DEBUG_FLAGS] =
      ((uint32_t)game->players[0].last_attack & 0xffU) |
      (((uint32_t)game->players[1].last_attack & 0xffU) << 8) |
      (((uint32_t)game->players[0].attack_phase & 0x0fU) << 16) |
      (((uint32_t)game->players[1].attack_phase & 0x0fU) << 20);
  regs[FIGHTER_MMIO_REG_PLAYER1_ATTACK_CMD] =
      (uint32_t)game->players[0].last_attack;
  regs[FIGHTER_MMIO_REG_PLAYER2_ATTACK_CMD] =
      (uint32_t)game->players[1].last_attack;
  regs[FIGHTER_MMIO_REG_PLAYER1_STATE_FRAME] =
      (uint32_t)game->players[0].state_frame;
  regs[FIGHTER_MMIO_REG_PLAYER2_STATE_FRAME] =
      (uint32_t)game->players[1].state_frame;
  regs[FIGHTER_MMIO_REG_PLAYER1_EVENT_FLAGS] =
      (uint32_t)game->players[0].event_flags;
```

```
  regs[FIGHTER_MMIO_REG_PLAYER2_EVENT_FLAGS] =
      (uint32_t)game->players[1].event_flags;
  regs[FIGHTER_MMIO_REG_PLAYER1_COMBAT_RESULT] =
      (uint32_t)game->players[0].combat_result;
  regs[FIGHTER_MMIO_REG_PLAYER2_COMBAT_RESULT] =
      (uint32_t)game->players[1].combat_result;
}
```

A.1.8 sw/render_if/fighter_renderer.c

```
#include "fighter_renderer.h"

#include "fighter_animation.h"
#include "fighter_vga_mmio.h"

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fcntl.h>
#include <linux/fb.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <unistd.h>

static const off_t k_fighter_vga_default_bridge_reset_addr = (off_t)0xFFD0501C;
static const off_t k_fighter_vga_default_mmio_addr = (off_t)0xFF240000;
static const uint32_t k_fighter_vga_ident = 0x56504741U;

typedef struct {
  char ch;
  unsigned char rows[7];
} fighter_glyph_t;

static const fighter_glyph_t k_fighter_glyphs[] = {
    {' ', {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}},
    {'0', {0x0e, 0x11, 0x13, 0x15, 0x19, 0x11, 0x0e}},
    {'1', {0x04, 0x0c, 0x04, 0x04, 0x04, 0x04, 0x0e}},
    {'2', {0x0e, 0x11, 0x01, 0x02, 0x04, 0x08, 0x1f}},
    {'3', {0x1e, 0x01, 0x01, 0x0e, 0x01, 0x01, 0x1e}},
    {'4', {0x02, 0x06, 0x0a, 0x12, 0x1f, 0x02, 0x02}},
    {'5', {0x1f, 0x10, 0x10, 0x1e, 0x01, 0x01, 0x1e}},
    {'6', {0x0e, 0x10, 0x10, 0x1e, 0x11, 0x11, 0x0e}},
    {'7', {0x1f, 0x01, 0x02, 0x04, 0x08, 0x08, 0x08}},
    {'8', {0x0e, 0x11, 0x11, 0x0e, 0x11, 0x11, 0x0e}},
    {'9', {0x0e, 0x11, 0x11, 0x0f, 0x01, 0x01, 0x0e}},
    {':', {0x00, 0x04, 0x04, 0x00, 0x04, 0x04, 0x00}},
    {'-', {0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00}},
    {'A', {0x0e, 0x11, 0x11, 0x1f, 0x11, 0x11, 0x11}},
    {'B', {0x1e, 0x11, 0x11, 0x1e, 0x11, 0x11, 0x1e}},
    {'C', {0x0e, 0x11, 0x10, 0x10, 0x10, 0x11, 0x0e}},
    {'D', {0x1c, 0x12, 0x11, 0x11, 0x11, 0x12, 0x1c}},
    {'E', {0x1f, 0x10, 0x10, 0x1e, 0x10, 0x10, 0x1f}},
    {'F', {0x1f, 0x10, 0x10, 0x1e, 0x10, 0x10, 0x10}},
    {'G', {0x0e, 0x11, 0x10, 0x10, 0x13, 0x11, 0x0f}},
    {'H', {0x11, 0x11, 0x11, 0x1f, 0x11, 0x11, 0x11}},
    {'I', {0x0e, 0x04, 0x04, 0x04, 0x04, 0x04, 0x0e}},
    {'J', {0x01, 0x01, 0x01, 0x01, 0x11, 0x11, 0x0e}},
    {'K', {0x11, 0x12, 0x14, 0x18, 0x14, 0x12, 0x11}},
    {'L', {0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1f}},
    {'M', {0x11, 0x1b, 0x15, 0x15, 0x11, 0x11, 0x11}},
    {'N', {0x11, 0x11, 0x19, 0x15, 0x13, 0x11, 0x11}},
    {'O', {0x0e, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0e}},
    {'P', {0x1e, 0x11, 0x11, 0x1e, 0x10, 0x10, 0x10}},
    {'Q', {0x0e, 0x11, 0x11, 0x11, 0x15, 0x12, 0x0d}},
    {'R', {0x1e, 0x11, 0x11, 0x1e, 0x14, 0x12, 0x11}},
    {'S', {0x0f, 0x10, 0x10, 0x0e, 0x01, 0x01, 0x1e}},
    {'T', {0x1f, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04}},
    {'U', {0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x0e}},
```

```
    {'V', {0x11, 0x11, 0x11, 0x11, 0x11, 0x0a, 0x04}},
    {'W', {0x11, 0x11, 0x11, 0x15, 0x15, 0x15, 0x0a}},
    {'X', {0x11, 0x11, 0x0a, 0x04, 0x0a, 0x11, 0x11}},
    {'Y', {0x11, 0x11, 0x0a, 0x04, 0x04, 0x04, 0x04}},
    {'Z', {0x1f, 0x01, 0x02, 0x04, 0x08, 0x10, 0x1f}},
};


static const fighter_glyph_t *fighter_find_glyph(char ch) {
  size_t i;

  if (ch >= 'a' && ch <= 'z') {
    ch = (char)(ch - 'a' + 'A');
  }

  for (i = 0; i < sizeof(k_fighter_glyphs) / sizeof(k_fighter_glyphs[0]); ++i) {
    if (k_fighter_glyphs[i].ch == ch) {
      return &k_fighter_glyphs[i];
    }
  }

  return &k_fighter_glyphs[0];
}

static void fighter_renderer_draw_menu_fb(fighter_renderer_t *renderer,
                                          const fighter_game_t *game);
static void fighter_renderer_draw_playfield_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system,
    int draw_overlay);
static void fighter_renderer_draw_game_over_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system);
static int fighter_rgb_image_load_rgb565(fighter_rgb_image_t *image,
                                         const char *path);
static int fighter_fb_image_build_scaled(fighter_renderer_t *renderer,
                                         const fighter_rgb_image_t *source,
                                         fighter_fb_image_t *scaled);
static int fighter_fb_image_build_cover(fighter_renderer_t *renderer,
                                        const fighter_rgb_image_t *source,
                                        fighter_fb_image_t *scaled);


const char *fighter_renderer_menu_frame_path(int frame_index) {
  static const char *const k_menu_frames[2] = {
      "../game_assets/ui/menu/menu_frame_0.png",
      "../game_assets/ui/menu/menu_frame_1.png",
  };

  if ((frame_index & 1) == 0) {
    return k_menu_frames[0];
  }
  return k_menu_frames[1];
}


static const char *fighter_renderer_menu_frame_rgb565_path(int frame_index) {
  static const char *const k_menu_frames[2] = {
      "../game_assets/ui/menu/menu_frame_0.rgb565",
      "../game_assets/ui/menu/menu_frame_1.rgb565",
  };

  if ((frame_index & 1) == 0) {
    return k_menu_frames[0];
  }
  return k_menu_frames[1];
}


static int fighter_renderer_parse_env_address(const char *env_name,
                                              off_t default_value,
```

```
                                            off_t *value_out) {
  const char *text;
  char *end;
  unsigned long long parsed;

  if (!env_name || !value_out) {
    return -1;
  }

  text = getenv(env_name);
  if (!text || text[0] == '\0') {
    *value_out = default_value;
    return 0;
  }

  errno = 0;
  parsed = strtoull(text, &end, 0);
  if (errno != 0 || end == text || !end || *end != '\0') {
    return -1;
  }

  *value_out = (off_t)parsed;
  return 0;
}


static int fighter_renderer_map_physical(int mem_fd,
                                         off_t physical_addr,
                                         size_t span,
                                         void **map_base,
                                         unsigned long *map_length,
                                         volatile uint32_t **register_base) {
  long page_size;
  off_t page_base;
  off_t page_offset;
  size_t length;
  void *mapped;

  if (mem_fd < 0 || !map_base || !map_length || !register_base || span == 0) {
    return -1;
  }

  page_size = sysconf(_SC_PAGESIZE);
  if (page_size <= 0) {
    return -1;
  }

  page_base = physical_addr & ~((off_t)page_size - 1);
  page_offset = physical_addr - page_base;
  length = (size_t)page_offset + span;
  length = (length + (size_t)page_size - 1U) & ~((size_t)page_size - 1U);

  mapped = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd,
                page_base);
  if (mapped == MAP_FAILED) {
    return -1;
  }

  *map_base = mapped;
  *map_length = (unsigned long)length;
  *register_base =
      (volatile uint32_t *)((unsigned char *)mapped + (size_t)page_offset);
  return 0;
}


static void fighter_renderer_unmap_region(void **map_base,
                                          unsigned long *map_length) {
  if (!map_base || !map_length || !*map_base || *map_length == 0) {
    return;
  }

  munmap(*map_base, (size_t)*map_length);
```

```
  *map_base = NULL;
  *map_length = 0;
}


static int fighter_renderer_enable_fpga_bridges(fighter_renderer_t *renderer) {
  uint32_t value;

  if (!renderer || !renderer->vga_bridge_reset_reg) {
    return -1;
  }

  value = *renderer->vga_bridge_reset_reg;
  value &= ~0x3U;
  *renderer->vga_bridge_reset_reg = value;
  return 0;
}


static int fighter_renderer_load_asset_rgb565(fighter_rgb_image_t *image,
                                              const char *relative_path) {
  static const char *const k_asset_roots[] = {
      NULL,
      "/root/game_assets",
      "../game_assets",
  };
  const char *env_root = getenv("FIGHTER_ASSET_ROOT");
  int root_index;

  if (!image || !relative_path) {
    return -1;
  }

  for (root_index = 0;
       root_index < (int)(sizeof(k_asset_roots) / sizeof(k_asset_roots[0]));
       ++root_index) {
    const char *root = root_index == 0 ? env_root : k_asset_roots[root_index];
    char path[512];

    if (!root || root[0] == '\0') {
      continue;
    }
    if (snprintf(path, sizeof(path), "%s/%s", root, relative_path) >=
        (int)sizeof(path)) {
      continue;
    }
    if (fighter_rgb_image_load_rgb565(image, path) == 0) {
      return 0;
    }
  }

  return -1;
}


static void fighter_renderer_load_assets(fighter_renderer_t *renderer) {
  int i;

  if (!renderer) {
    return;
  }

  for (i = 0; i < 2; ++i) {
    char relative_path[64];

    snprintf(relative_path, sizeof(relative_path), "ui/menu/menu_frame_%d.rgb565", i);
    (void)fighter_renderer_load_asset_rgb565(&renderer->menu_frames[i],
                                             relative_path);
    if (!renderer->menu_frames[i].pixels) {
      (void)fighter_rgb_image_load_rgb565(
          &renderer->menu_frames[i], fighter_renderer_menu_frame_rgb565_path(i));
    }
    if (renderer->menu_frames[i].pixels) {
```

```
      (void)fighter_fb_image_build_scaled(renderer, &renderer->menu_frames[i],
                                          &renderer->menu_frame_cache[i]);
    }
  }

  if (fighter_renderer_load_asset_rgb565(&renderer->background_image,
                                         "background/background.rgb565") == 0) {
    (void)fighter_fb_image_build_cover(renderer, &renderer->background_image,
                                       &renderer->background_cache);
  }
}


static void fighter_renderer_close_mmio(fighter_renderer_t *renderer) {
  if (!renderer) {
    return;
  }

  fighter_renderer_unmap_region(&renderer->vga_regs_map,
                                &renderer->vga_regs_map_length);
  fighter_renderer_unmap_region(&renderer->vga_bridge_map,
                                &renderer->vga_bridge_map_length);
  if (renderer->vga_mem_fd >= 0) {
    close(renderer->vga_mem_fd);
    renderer->vga_mem_fd = -1;
  }
  renderer->vga_regs = NULL;
  renderer->vga_bridge_reset_reg = NULL;
}


static int fighter_renderer_prepare_mmio_framebuffer(
    fighter_renderer_t *renderer) {
  if (!renderer) {
    return -1;
  }

  renderer->fb_width = FIGHTER_VGA_MMIO_FRAME_WIDTH;
  renderer->fb_height = FIGHTER_VGA_MMIO_FRAME_HEIGHT;
  renderer->fb_stride = FIGHTER_VGA_MMIO_FRAME_WIDTH * 2;
  renderer->fb_bpp = 16;
  renderer->fb_data_length =
      (unsigned long)(renderer->fb_stride * renderer->fb_height);
  renderer->fb_backbuffer = (unsigned char *)malloc(renderer->fb_data_length);
  renderer->fb_backbuffer_length = renderer->fb_data_length;
  if (!renderer->fb_backbuffer) {
    renderer->fb_backbuffer_length = 0;
    return -1;
  }

  memset(renderer->fb_backbuffer, 0, renderer->fb_backbuffer_length);
  return 0;
}


static int fighter_renderer_init_mmio(fighter_renderer_t *renderer) {
  off_t bridge_reset_addr;
  off_t mmio_addr;
  uint32_t ident;

  if (!renderer) {
    return -1;
  }

  renderer->vga_mem_fd = -1;
  if (fighter_renderer_parse_env_address("FIGHTER_VGA_BRIDGE_RESET_ADDR",
                                         k_fighter_vga_default_bridge_reset_addr,
                                         &bridge_reset_addr) != 0) {
    snprintf(renderer->init_status, sizeof(renderer->init_status),
             "VGA MMIO: invalid FIGHTER_VGA_BRIDGE_RESET_ADDR=%s",
             getenv("FIGHTER_VGA_BRIDGE_RESET_ADDR"));
    return -1;
  }
```

```
353   if (fighter_renderer_parse_env_address("FIGHTER_VGA_MMIO_ADDR",
354                                          k_fighter_vga_default_mmio_addr,
355                                          &mmio_addr) != 0) {
356     snprintf(renderer->init_status, sizeof(renderer->init_status),
357              "VGA MMIO: invalid FIGHTER_VGA_MMIO_ADDR=%s",
358              getenv("FIGHTER_VGA_MMIO_ADDR"));
359     return -1;
360   }
361
362   renderer->vga_mmio_addr = (unsigned long)mmio_addr;
363   renderer->vga_bridge_reset_addr = (unsigned long)bridge_reset_addr;
364   renderer->vga_mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
365   if (renderer->vga_mem_fd < 0) {
366     snprintf(renderer->init_status, sizeof(renderer->init_status),
367              "VGA MMIO: open /dev/mem failed: %s", strerror(errno));
368     return -1;
369   }
370
371   if (fighter_renderer_map_physical(renderer->vga_mem_fd, bridge_reset_addr,
372                                     sizeof(uint32_t), &renderer->vga_bridge_map,
373                                     &renderer->vga_bridge_map_length,
374                                     &renderer->vga_bridge_reset_reg) != 0) {
375     snprintf(renderer->init_status, sizeof(renderer->init_status),
376              "VGA MMIO: map bridge reset 0x%08lX failed: %s",
377              (unsigned long)bridge_reset_addr, strerror(errno));
378     fighter_renderer_close_mmio(renderer);
379     return -1;
380   }
381   if (fighter_renderer_enable_fpga_bridges(renderer) != 0) {
382     snprintf(renderer->init_status, sizeof(renderer->init_status),
383              "VGA MMIO: failed to enable FPGA bridges");
384     fighter_renderer_close_mmio(renderer);
385     return -1;
386   }
387
388   if (fighter_renderer_map_physical(renderer->vga_mem_fd, mmio_addr,
389                                     FIGHTER_VGA_MMIO_REG_SPAN_COUNT *
390                                         sizeof(uint32_t),
391                                     &renderer->vga_regs_map,
392                                     &renderer->vga_regs_map_length,
393                                     &renderer->vga_regs) != 0) {
394     snprintf(renderer->init_status, sizeof(renderer->init_status),
395              "VGA MMIO: map renderer core 0x%08lX failed: %s",
396              (unsigned long)mmio_addr, strerror(errno));
397     fighter_renderer_close_mmio(renderer);
398     return -1;
399   }
400
401   ident = renderer->vga_regs[FIGHTER_VGA_MMIO_REG_IDENT];
402   if (ident != k_fighter_vga_ident) {
403     snprintf(renderer->init_status, sizeof(renderer->init_status),
404              "VGA MMIO: probe failed at 0x%08lX (ident=0x%08X)",
405              (unsigned long)mmio_addr, ident);
406     fighter_renderer_close_mmio(renderer);
407     return -1;
408   }
409
410   if (renderer->vga_regs[FIGHTER_VGA_MMIO_REG_WIDTH] !=
411           FIGHTER_VGA_MMIO_FRAME_WIDTH ||
412       renderer->vga_regs[FIGHTER_VGA_MMIO_REG_HEIGHT] !=
413           FIGHTER_VGA_MMIO_FRAME_HEIGHT ||
414       renderer->vga_regs[FIGHTER_VGA_MMIO_REG_STRIDE] !=
415           FIGHTER_VGA_MMIO_FRAME_WIDTH * 2) {
416     snprintf(renderer->init_status, sizeof(renderer->init_status),
417              "VGA MMIO: framebuffer geometry mismatch (%ux%u stride=%u)",
418              renderer->vga_regs[1], renderer->vga_regs[2], renderer->vga_regs[3]);
419     fighter_renderer_close_mmio(renderer);
420     return -1;
421   }
422
423   if (fighter_renderer_prepare_mmio_framebuffer(renderer) != 0) {
424     snprintf(renderer->init_status, sizeof(renderer->init_status),
425              "VGA MMIO: failed to allocate %dx%d backbuffer",
```

```
426             FIGHTER_VGA_MMIO_FRAME_WIDTH, FIGHTER_VGA_MMIO_FRAME_HEIGHT);
427     fighter_renderer_close_mmio(renderer);
428     return -1;
429   }
430
431   fighter_renderer_load_assets(renderer);
432   renderer->backend = FIGHTER_RENDERER_BACKEND_MMIO;
433   snprintf(renderer->init_status, sizeof(renderer->init_status),
434            "VGA MMIO framebuffer active at 0x%08lX (%dx%d RGB565)",
435            (unsigned long)mmio_addr, renderer->fb_width, renderer->fb_height);
436   return 0;
437 }
438
439
440 static void fighter_renderer_flush_mmio_frame(fighter_renderer_t *renderer) {
441   const unsigned char *src;
442   volatile uint32_t *dst;
443   int word_count;
444   int i;
445   int wait_count;
446
447   if (!renderer || !renderer->vga_regs || !renderer->fb_backbuffer) {
448     return;
449   }
450
451   for (wait_count = 0; wait_count < 10000000; ++wait_count) {
452     if ((renderer->vga_regs[FIGHTER_VGA_MMIO_REG_CONTROL] &
453          FIGHTER_VGA_MMIO_CONTROL_SWAP_PENDING) == 0U) {
454       break;
455     }
456   }
457
458   src = renderer->fb_backbuffer;
459   dst = renderer->vga_regs + FIGHTER_VGA_MMIO_REG_FRAME_WORD_OFFSET;
460   word_count = FIGHTER_VGA_MMIO_FRAME_WORD_COUNT;
461   for (i = 0; i < word_count; ++i) {
462     uint32_t lo =
463         (uint32_t)src[(size_t)i * 4U] |
464         ((uint32_t)src[(size_t)i * 4U + 1U] << 8);
465     uint32_t hi =
466         (uint32_t)src[(size_t)i * 4U + 2U] |
467         ((uint32_t)src[(size_t)i * 4U + 3U] << 8);
468     dst[i] = lo | (hi << 16);
469   }
470
471   renderer->vga_regs[FIGHTER_VGA_MMIO_REG_CONTROL] =
472       FIGHTER_VGA_MMIO_CONTROL_SWAP_REQUEST;
473 }
474
475
476 static void fighter_renderer_draw_mmio(
477     fighter_renderer_t *renderer,
478     const fighter_game_t *game,
479     const fighter_animation_system_t *anim_system) {
480   if (!renderer || !renderer->vga_regs || !game) {
481     return;
482   }
483
484   switch (game->state) {
485     case FIGHTER_GAME_STATE_MENU:
486       fighter_renderer_draw_menu_fb(renderer, game);
487       break;
488     case FIGHTER_GAME_STATE_PLAYING:
489       fighter_renderer_draw_playfield_fb(renderer, game, anim_system, 0);
490       break;
491     case FIGHTER_GAME_STATE_GAME_OVER:
492       fighter_renderer_draw_game_over_fb(renderer, game, anim_system);
493       break;
494     default:
495       break;
496   }
497   fighter_renderer_flush_mmio_frame(renderer);
498 }
```

```
499
500
501 void fighter_renderer_options_init(fighter_renderer_options_t *options) {
502   if (!options) {
503     return;
504   }
505
506   memset(options, 0, sizeof(*options));
507   options->prefer_framebuffer = 1;
508   options->console_interval_frames = 15;
509   options->framebuffer_path = "/dev/fb0";
510 }
511
512
513 static const char *fighter_renderer_game_state_name(fighter_game_state_t state) {
514   switch (state) {
515     case FIGHTER_GAME_STATE_MENU:
516       return "MENU";
517     case FIGHTER_GAME_STATE_PLAYING:
518       return "PLAYING";
519     case FIGHTER_GAME_STATE_GAME_OVER:
520       return "GAME_OVER";
521     default:
522       return "UNKNOWN";
523   }
524 }
525
526
527 static const char *fighter_renderer_visual_state_name(
528     fighter_visual_state_t state) {
529   switch (state) {
530     case FIGHTER_VISUAL_STATE_IDLE:
531       return "IDLE";
532     case FIGHTER_VISUAL_STATE_WALK:
533       return "WALK";
534     case FIGHTER_VISUAL_STATE_JUMP:
535       return "JUMP";
536     case FIGHTER_VISUAL_STATE_CROUCH:
537       return "CROUCH";
538     case FIGHTER_VISUAL_STATE_GUARD:
539       return "STAND_GUARD";
540     case FIGHTER_VISUAL_STATE_CROUCH_GUARD:
541       return "CROUCH_GUARD";
542     case FIGHTER_VISUAL_STATE_ATTACK:
543       return "ATTACK";
544     case FIGHTER_VISUAL_STATE_HIT:
545       return "HIT";
546     case FIGHTER_VISUAL_STATE_BLOCK_STUN:
547       return "BLOCK_STUN";
548     case FIGHTER_VISUAL_STATE_KO:
549       return "KO";
550     case FIGHTER_VISUAL_STATE_VICTORY:
551       return "VICTORY";
552     default:
553       return "UNKNOWN";
554   }
555 }
556
557
558 static const char *fighter_renderer_attack_phase_name(
559     fighter_attack_phase_t phase) {
560   switch (phase) {
561     case FIGHTER_ATTACK_PHASE_NONE:
562       return "NONE";
563     case FIGHTER_ATTACK_PHASE_STARTUP:
564       return "STARTUP";
565     case FIGHTER_ATTACK_PHASE_ACTIVE:
566       return "ACTIVE";
567     case FIGHTER_ATTACK_PHASE_HIT_CONFIRM:
568       return "HIT_CONFIRM";
569     case FIGHTER_ATTACK_PHASE_BLOCK_CONFIRM:
570       return "BLOCK_CONFIRM";
571     case FIGHTER_ATTACK_PHASE_RECOVERY:
```

```
572       return "RECOVERY";
573     default:
574       return "UNKNOWN";
575   }
576 }
577
578
579 static const char *fighter_renderer_winner_name(fighter_winner_t winner) {
580   switch (winner) {
581     case FIGHTER_WINNER_PLAYER1:
582       return "P1";
583     case FIGHTER_WINNER_PLAYER2:
584       return "P2";
585     case FIGHTER_WINNER_DRAW:
586       return "DRAW";
587     case FIGHTER_WINNER_NONE:
588     default:
589       return "NONE";
590   }
591 }
592
593
594 static const char *fighter_renderer_finish_reason_name(
595     fighter_finish_reason_t reason) {
596   switch (reason) {
597     case FIGHTER_FINISH_REASON_NONE:
598       return "NONE";
599     case FIGHTER_FINISH_REASON_KO:
600       return "KO";
601     case FIGHTER_FINISH_REASON_TIME_OUT:
602       return "TIME_OUT";
603     case FIGHTER_FINISH_REASON_DOUBLE_KO:
604       return "DOUBLE_KO";
605     case FIGHTER_FINISH_REASON_EXIT:
606       return "EXIT";
607     default:
608       return "UNKNOWN";
609   }
610 }
611
612
613 static int fighter_renderer_console_player_changed(
614     const fighter_player_state_t *lhs,
615     const fighter_player_state_t *rhs) {
616   return memcmp(lhs, rhs, sizeof(*lhs)) != 0;
617 }
618
619
620 static void fighter_renderer_print_console_player(
621     const char *label,
622     const fighter_player_state_t *player) {
623   if (!label || !player) {
624     return;
625   }
626
627   printf("  %s x=%d y=%d hp=%d face=%d vis=%s atk=%s phase=%s state_frame=%u\n",
628          label, player->x, player->y, player->hp, player->facing,
629          fighter_renderer_visual_state_name(player->visual_state),
630          fighter_attack_command_name(player->last_attack),
631          fighter_renderer_attack_phase_name(player->attack_phase),
632          player->state_frame);
633 }
634
635
636 static unsigned char *fighter_fb_target_data(fighter_renderer_t *renderer) {
637   if (!renderer) {
638     return NULL;
639   }
640   if (renderer->fb_backbuffer) {
641     return renderer->fb_backbuffer;
642   }
643   return (unsigned char *)renderer->fb_data;
644 }
```

```
645
646
647 static unsigned int fighter_fb_color(fighter_renderer_t *renderer,
648                                      unsigned char r,
649                                      unsigned char g,
650                                      unsigned char b) {
651   if (!renderer) {
652     return 0;
653   }
654
655   if (renderer->fb_bpp == 16) {
656     unsigned int rr = (unsigned int)(r >> 3);
657     unsigned int gg = (unsigned int)(g >> 2);
658     unsigned int bb = (unsigned int)(b >> 3);
659     return (rr << 11) | (gg << 5) | bb;
660   }
661
662   return ((unsigned int)r << 16) | ((unsigned int)g << 8) | (unsigned int)b;
663 }
664
665
666 static unsigned int fighter_rgb565_to_fb_color(fighter_renderer_t *renderer,
667                                                unsigned int rgb565) {
668   unsigned char r;
669   unsigned char g;
670   unsigned char b;
671
672   /* MMIO and 16bpp framebuffer paths both use little-endian RGB565. */
673   if (renderer && renderer->fb_bpp == 16) {
674     return rgb565;
675   }
676
677
678   r = (unsigned char)(((rgb565 >> 11) & 0x1fU) << 3);
679   g = (unsigned char)(((rgb565 >> 5) & 0x3fU) << 2);
680   b = (unsigned char)((rgb565 & 0x1fU) << 3);
681   r = (unsigned char)(r | (r >> 5));
682   g = (unsigned char)(g | (g >> 6));
683   b = (unsigned char)(b | (b >> 5));
684   return fighter_fb_color(renderer, r, g, b);
685 }
686
687
688 static void fighter_fb_store_color(fighter_renderer_t *renderer,
689                                    unsigned char *dst,
690                                    unsigned int color) {
691   if (!renderer || !dst) {
692     return;
693   }
694
695   if (renderer->fb_bpp == 16) {
696     dst[0] = (unsigned char)(color & 0xff);
697     dst[1] = (unsigned char)((color >> 8) & 0xff);
698   } else {
699     dst[0] = (unsigned char)(color & 0xff);
700     dst[1] = (unsigned char)((color >> 8) & 0xff);
701     dst[2] = (unsigned char)((color >> 16) & 0xff);
702     if (renderer->fb_bpp >= 32) {
703       dst[3] = 0;
704     }
705   }
706 }
707
708
709 static void fighter_fb_put_pixel(fighter_renderer_t *renderer,
710                                  int x,
711                                  int y,
712                                  unsigned int color) {
713   unsigned char *target;
714   int bytes_per_pixel;
715   unsigned char *dst;
716
717   if (!renderer) {
```

```
    return;
  }
  if (x < 0 || y < 0 || x >= renderer->fb_width || y >= renderer->fb_height) {
    return;
  }

  target = fighter_fb_target_data(renderer);
  if (!target) {
    return;
  }

  bytes_per_pixel = renderer->fb_bpp / 8;
  if (bytes_per_pixel <= 0) {
    return;
  }

  dst = target + (size_t)y * (size_t)renderer->fb_stride +
        (size_t)x * (size_t)bytes_per_pixel;
  fighter_fb_store_color(renderer, dst, color);
}


static int fighter_scale_axis(int value, int dst_extent, int src_extent) {
  if (src_extent <= 0) {
    return 0;
  }
  return (int)(((long long)value * dst_extent) / src_extent);
}


static int fighter_scale_size_axis(int value, int dst_extent, int src_extent) {
  int out = fighter_scale_axis(value, dst_extent, src_extent);
  return out > 0 ? out : 1;
}


static int fighter_scale_text_size(fighter_renderer_t *renderer, int base_scale) {
  int s;

  if (!renderer || base_scale <= 0) {
    return 1;
  }

  s = fighter_scale_size_axis(base_scale, renderer->fb_height, 480);
  return s > 0 ? s : 1;
}


static void fighter_fb_fill_rect(fighter_renderer_t *renderer,
                                 int x,
                                 int y,
                                 int w,
                                 int h,
                                 unsigned int color) {
  unsigned char *target;
  int bytes_per_pixel;
  int yy;
  int xx;

  if (!renderer || w <= 0 || h <= 0) {
    return;
  }

  target = fighter_fb_target_data(renderer);
  if (!target) {
    return;
  }

  bytes_per_pixel = renderer->fb_bpp / 8;
  if (bytes_per_pixel <= 0) {
    return;
  }

```

```
  if (x < 0) {
    w += x;
    x = 0;
  }
  if (y < 0) {
    h += y;
    y = 0;
  }
  if (x + w > renderer->fb_width) {
    w = renderer->fb_width - x;
  }
  if (y + h > renderer->fb_height) {
    h = renderer->fb_height - y;
  }
  if (w <= 0 || h <= 0) {
    return;
  }

  for (yy = 0; yy < h; ++yy) {
    unsigned char *row = target + (size_t)(y + yy) * (size_t)renderer->fb_stride +
                         (size_t)x * (size_t)bytes_per_pixel;
    for (xx = 0; xx < w; ++xx) {
      fighter_fb_store_color(renderer, row + (size_t)xx * (size_t)bytes_per_pixel,
                             color);
    }
  }
}


static void fighter_fb_draw_char(fighter_renderer_t *renderer,
                                 int x,
                                 int y,
                                 char ch,
                                 int scale,
                                 unsigned int color) {
  const fighter_glyph_t *glyph;
  int row;
  int col;

  if (!renderer || scale <= 0) {
    return;
  }

  glyph = fighter_find_glyph(ch);
  for (row = 0; row < 7; ++row) {
    for (col = 0; col < 5; ++col) {
      if ((glyph->rows[row] >> (4 - col)) & 1U) {
        fighter_fb_fill_rect(renderer, x + col * scale, y + row * scale, scale,
                             scale, color);
      }
    }
  }
}


static void fighter_fb_draw_text(fighter_renderer_t *renderer,
                                 int x,
                                 int y,
                                 const char *text,
                                 int scale,
                                 unsigned int color) {
  int cursor_x;
  size_t i;

  if (!renderer || !text || scale <= 0) {
    return;
  }

  cursor_x = x;
  for (i = 0; text[i] != '\0'; ++i) {
    fighter_fb_draw_char(renderer, cursor_x, y, text[i], scale, color);
    cursor_x += 6 * scale;
  }
```

```
}


static void fighter_fb_draw_centered_text(fighter_renderer_t *renderer,
                                          int center_x,
                                          int y,
                                          const char *text,
                                          int scale,
                                          unsigned int color) {
  int text_width;

  if (!renderer || !text || scale <= 0) {
    return;
  }

  text_width = (int)strlen(text) * 6 * scale - scale;
  fighter_fb_draw_text(renderer, center_x - text_width / 2, y, text, scale, color);
}


static void fighter_rgb_image_reset(fighter_rgb_image_t *image) {
  if (!image) {
    return;
  }

  free(image->pixels);
  image->pixels = NULL;
  image->width = 0;
  image->height = 0;
}


static void fighter_fb_image_reset(fighter_fb_image_t *image) {
  if (!image) {
    return;
  }

  free(image->pixels);
  image->pixels = NULL;
  image->width = 0;
  image->height = 0;
  image->stride = 0;
  image->data_length = 0;
}


static unsigned int fighter_read_le16(const unsigned char *data) {
  return (unsigned int)data[0] | ((unsigned int)data[1] << 8);
}


static unsigned long fighter_read_le32(const unsigned char *data) {
  return (unsigned long)data[0] | ((unsigned long)data[1] << 8) |
         ((unsigned long)data[2] << 16) | ((unsigned long)data[3] << 24);
}


static int fighter_rgb_image_load_rgb565(fighter_rgb_image_t *image,
                                         const char *path) {
  FILE *stream;
  unsigned char header[16];
  int width;
  int height;
  unsigned long data_size;
  size_t bytes_needed;
  unsigned char *pixels;

  if (!image || !path) {
    return -1;
  }

  stream = fopen(path, "rb");
  if (!stream) {
```

```
    return -1;
  }

  if (fread(header, 1, sizeof(header), stream) != sizeof(header)) {
    fclose(stream);
    return -1;
  }

  if (memcmp(header, "R565", 4) != 0 || fighter_read_le16(header + 4) != 16 ||
      fighter_read_le16(header + 10) != 1) {
    fclose(stream);
    return -1;
  }

  width = (int)fighter_read_le16(header + 6);
  height = (int)fighter_read_le16(header + 8);
  data_size = fighter_read_le32(header + 12);
  if (width <= 0 || height <= 0) {
    fclose(stream);
    return -1;
  }

  bytes_needed = (size_t)width * (size_t)height * 2U;
  if (data_size != (unsigned long)bytes_needed) {
    fclose(stream);
    return -1;
  }

  pixels = (unsigned char *)malloc(bytes_needed);
  if (!pixels) {
    fclose(stream);
    return -1;
  }

  if (fread(pixels, 1, bytes_needed, stream) != bytes_needed) {
    free(pixels);
    fclose(stream);
    return -1;
  }

  fclose(stream);
  fighter_rgb_image_reset(image);
  image->width = width;
  image->height = height;
  image->pixels = pixels;
  return 0;
}


static void fighter_fb_draw_rgb_image_fit(fighter_renderer_t *renderer,
                                          const fighter_rgb_image_t *image) {
  int draw_width;
  int draw_height;
  int draw_x;
  int draw_y;
  int y;

  if (!renderer || !image || !image->pixels || image->width <= 0 || image->height <= 0) {
    return;
  }

  draw_width = renderer->fb_width;
  draw_height = (int)(((long long)draw_width * image->height) / image->width);
  if (draw_height > renderer->fb_height) {
    draw_height = renderer->fb_height;
    draw_width = (int)(((long long)draw_height * image->width) / image->height);
  }
  if (draw_width <= 0 || draw_height <= 0) {
    return;
  }

  draw_x = (renderer->fb_width - draw_width) / 2;
  draw_y = (renderer->fb_height - draw_height) / 2;
```

```
  fighter_fb_fill_rect(renderer, 0, 0, renderer->fb_width, renderer->fb_height,
                       fighter_fb_color(renderer, 0, 0, 0));

  for (y = 0; y < draw_height; ++y) {
    int src_y = (int)(((long long)y * image->height) / draw_height);
    const unsigned char *src_row =
        image->pixels + (size_t)src_y * (size_t)image->width * 2U;
    int x;

    for (x = 0; x < draw_width; ++x) {
      int src_x = (int)(((long long)x * image->width) / draw_width);
      const unsigned char *src_pixel = src_row + (size_t)src_x * 2U;
      fighter_fb_put_pixel(renderer, draw_x + x, draw_y + y,
                           fighter_rgb565_to_fb_color(
                               renderer, fighter_read_le16(src_pixel)));
    }
  }
}


static int fighter_fb_image_build_scaled(fighter_renderer_t *renderer,
                                         const fighter_rgb_image_t *source,
                                         fighter_fb_image_t *scaled) {
  int draw_width;
  int draw_height;
  int draw_x;
  int draw_y;
  int bytes_per_pixel;
  int y;

  if (!renderer || !source || !source->pixels || !scaled || renderer->fb_width <= 0 ||
      renderer->fb_height <= 0 || renderer->fb_stride <= 0) {
    return -1;
  }

  fighter_fb_image_reset(scaled);

  bytes_per_pixel = renderer->fb_bpp / 8;
  if (bytes_per_pixel <= 0) {
    return -1;
  }

  scaled->data_length = (unsigned long)(renderer->fb_stride * renderer->fb_height);
  scaled->pixels = (unsigned char *)malloc(scaled->data_length);
  if (!scaled->pixels) {
    fighter_fb_image_reset(scaled);
    return -1;
  }

  scaled->width = renderer->fb_width;
  scaled->height = renderer->fb_height;
  scaled->stride = renderer->fb_stride;
  memset(scaled->pixels, 0, scaled->data_length);

  draw_width = renderer->fb_width;
  draw_height = (int)(((long long)draw_width * source->height) / source->width);
  if (draw_height > renderer->fb_height) {
    draw_height = renderer->fb_height;
    draw_width = (int)(((long long)draw_height * source->width) / source->height);
  }
  if (draw_width <= 0 || draw_height <= 0) {
    fighter_fb_image_reset(scaled);
    return -1;
  }

  draw_x = (renderer->fb_width - draw_width) / 2;
  draw_y = (renderer->fb_height - draw_height) / 2;

  for (y = 0; y < draw_height; ++y) {
    int src_y = (int)(((long long)y * source->height) / draw_height);
    const unsigned char *src_row =
        source->pixels + (size_t)src_y * (size_t)source->width * 2U;
    int x;
```

```
1083
1084      for (x = 0; x < draw_width; ++x) {
1085        int src_x = (int)(((long long)x * source->width) / draw_width);
1086        const unsigned char *src_pixel = src_row + (size_t)src_x * 2U;
1087        unsigned int color =
1088            fighter_rgb565_to_fb_color(renderer, fighter_read_le16(src_pixel));
1089        unsigned char *dst =
1090            scaled->pixels + (size_t)(draw_y + y) * (size_t)scaled->stride +
1091            (size_t)(draw_x + x) * (size_t)bytes_per_pixel;
1092        fighter_fb_store_color(renderer, dst, color);
1093      }
1094    }
1095
1096    return 0;
1097  }
1098
1099
1100  static int fighter_fb_image_build_cover(fighter_renderer_t *renderer,
1101                                          const fighter_rgb_image_t *source,
1102                                          fighter_fb_image_t *scaled) {
1103    int bytes_per_pixel;
1104    int crop_x;
1105    int crop_y;
1106    int crop_w;
1107    int crop_h;
1108    int y;
1109
1110    if (!renderer || !source || !source->pixels || !scaled ||
1111        renderer->fb_width <= 0 || renderer->fb_height <= 0 ||
1112        renderer->fb_stride <= 0) {
1113      return -1;
1114    }
1115
1116    fighter_fb_image_reset(scaled);
1117
1118    bytes_per_pixel = renderer->fb_bpp / 8;
1119    if (bytes_per_pixel <= 0) {
1120      return -1;
1121    }
1122
1123    scaled->data_length = (unsigned long)(renderer->fb_stride * renderer->fb_height);
1124    scaled->pixels = (unsigned char *)malloc(scaled->data_length);
1125    if (!scaled->pixels) {
1126      fighter_fb_image_reset(scaled);
1127      return -1;
1128    }
1129
1130    scaled->width = renderer->fb_width;
1131    scaled->height = renderer->fb_height;
1132    scaled->stride = renderer->fb_stride;
1133    memset(scaled->pixels, 0, scaled->data_length);
1134
1135    crop_x = 0;
1136    crop_y = 0;
1137    crop_w = source->width;
1138    crop_h = source->height;
1139
1140    if ((long long)source->width * renderer->fb_height >
1141        (long long)renderer->fb_width * source->height) {
1142      crop_w = (int)(((long long)source->height * renderer->fb_width) /
1143                     renderer->fb_height);
1144      if (crop_w <= 0) {
1145        crop_w = 1;
1146      }
1147      crop_x = (source->width - crop_w) / 2;
1148    } else {
1149      crop_h = (int)(((long long)source->width * renderer->fb_height) /
1150                     renderer->fb_width);
1151      if (crop_h <= 0) {
1152        crop_h = 1;
1153      }
1154      crop_y = (source->height - crop_h) / 2;
1155    }
```

```

  for (y = 0; y < renderer->fb_height; ++y) {
    int src_y = crop_y + (int)(((long long)y * crop_h) / renderer->fb_height);
    const unsigned char *src_row =
        source->pixels + (size_t)src_y * (size_t)source->width * 2U;
    int x;

    for (x = 0; x < renderer->fb_width; ++x) {
      int src_x = crop_x + (int)(((long long)x * crop_w) / renderer->fb_width);
      const unsigned char *src_pixel = src_row + (size_t)src_x * 2U;
      unsigned int color =
          fighter_rgb565_to_fb_color(renderer, fighter_read_le16(src_pixel));
      unsigned char *dst =
          scaled->pixels + (size_t)y * (size_t)scaled->stride +
          (size_t)x * (size_t)bytes_per_pixel;
      fighter_fb_store_color(renderer, dst, color);
    }
  }

  return 0;
}


static void fighter_fb_draw_cached_image(fighter_renderer_t *renderer,
                                         const fighter_fb_image_t *image) {
  if (!renderer || !image || !image->pixels) {
    return;
  }

  if (renderer->fb_backbuffer &&
      image->data_length == renderer->fb_backbuffer_length) {
    memcpy(renderer->fb_backbuffer, image->pixels, image->data_length);
    return;
  }

  if (renderer->fb_data && image->data_length == renderer->fb_data_length) {
    memcpy(renderer->fb_data, image->pixels, image->data_length);
  }
}


static void fighter_fb_present(fighter_renderer_t *renderer) {
  if (!renderer || !renderer->fb_backbuffer || !renderer->fb_data) {
    return;
  }

  memcpy(renderer->fb_data, renderer->fb_backbuffer, renderer->fb_backbuffer_length);
}


static void fighter_fb_clear(fighter_renderer_t *renderer, unsigned int color) {
  fighter_fb_fill_rect(renderer, 0, 0, renderer->fb_width, renderer->fb_height,
                       color);
}


static void fighter_fb_draw_hp_bar(fighter_renderer_t *renderer,
                                   int x,
                                   int y,
                                   int w,
                                   int h,
                                   int hp,
                                   int max_hp,
                                   unsigned int fg,
                                   unsigned int bg,
                                   unsigned int border) {
  int fill_w;

  fighter_fb_fill_rect(renderer, x, y, w, h, border);
  fighter_fb_fill_rect(renderer, x + 2, y + 2, w - 4, h - 4, bg);

  if (max_hp <= 0) {
    return;
```

```
  }

  fill_w = ((w - 4) * hp) / max_hp;
  if (fill_w < 0) {
    fill_w = 0;
  }
  if (fill_w > w - 4) {
    fill_w = w - 4;
  }

  fighter_fb_fill_rect(renderer, x + 2, y + 2, fill_w, h - 4, fg);
}


static void fighter_fb_draw_sprite(fighter_renderer_t *renderer,
                                   const fighter_sprite_t *sprite,
                                   int dst_x,
                                   int dst_y,
                                   int dst_w,
                                   int dst_h,
                                   int flip_x) {
  unsigned char *target;
  int bytes_per_pixel;
  int x;
  int y;

  if (!renderer || !sprite || !sprite->pixels || sprite->width <= 0 ||
      sprite->height <= 0 || dst_w <= 0 || dst_h <= 0) {
    return;
  }

  target = fighter_fb_target_data(renderer);
  if (!target) {
    return;
  }

  bytes_per_pixel = renderer->fb_bpp / 8;
  if (bytes_per_pixel <= 0) {
    return;
  }

  for (y = 0; y < dst_h; ++y) {
    int src_y = (int)(((long long)y * sprite->height) / dst_h);
    int py = dst_y + y;

    if (py < 0 || py >= renderer->fb_height) {
      continue;
    }

    for (x = 0; x < dst_w; ++x) {
      int src_x;
      int px = dst_x + x;
      const unsigned char *src_pixel;
      unsigned int rgb565;
      unsigned char *dst_pixel;

      if (px < 0 || px >= renderer->fb_width) {
        continue;
      }

      if (flip_x) {
        src_x = (int)(((long long)(dst_w - 1 - x) * sprite->width) / dst_w);
      } else {
        src_x = (int)(((long long)x * sprite->width) / dst_w);
      }

      src_pixel =
          sprite->pixels +
          ((size_t)src_y * (size_t)sprite->width + (size_t)src_x) * 2U;
      rgb565 = fighter_read_le16(src_pixel);


      if (rgb565 == 0U) {
```

```
        continue;
      }

      dst_pixel = target + (size_t)py * (size_t)renderer->fb_stride +
                  (size_t)px * (size_t)bytes_per_pixel;
      fighter_fb_store_color(renderer, dst_pixel,
                             fighter_rgb565_to_fb_color(renderer, rgb565));
    }
  }
}


static void fighter_renderer_draw_player_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system,
    int player_index) {
  const fighter_player_state_t *player;
  const fighter_sprite_t *sprite;
  int hitbox_x;
  int hitbox_y;
  int hitbox_w;
  int hitbox_h;
  int draw_w;
  int draw_h;
  int draw_x;
  int draw_y;
  int flip_x;

  if (!renderer || !game || !anim_system || player_index < 0 ||
      player_index >= FIGHTER_PLAYER_COUNT) {
    return;
  }

  player = &game->players[player_index];
  sprite = fighter_animation_current_sprite(anim_system, player_index);
  if (!sprite || !sprite->pixels || sprite->width <= 0 || sprite->height <= 0) {
    return;
  }


  hitbox_x =
      fighter_scale_axis(player->x, renderer->fb_width, game->config.screen_width);
  hitbox_y =
      fighter_scale_axis(player->y, renderer->fb_height, game->config.screen_height);
  hitbox_w =
      fighter_scale_size_axis(game->config.player_width, renderer->fb_width,
                              game->config.screen_width);
  hitbox_h =
      fighter_scale_size_axis(game->config.player_height, renderer->fb_height,
                              game->config.screen_height);


  draw_w =
      fighter_scale_size_axis(sprite->width, renderer->fb_width, game->config.screen_width);
  draw_h =
      fighter_scale_size_axis(sprite->height, renderer->fb_height, game->config.screen_height);


  draw_x = hitbox_x + (hitbox_w - draw_w) / 2;
  draw_y = hitbox_y + hitbox_h - draw_h;

  flip_x = (player->facing < 0);
  fighter_fb_draw_sprite(renderer, sprite, draw_x, draw_y, draw_w, draw_h, flip_x);
}


static const fighter_sprite_t *fighter_renderer_fireball_sprite(
    const fighter_animation_system_t *anim_system,
    fighter_character_id_t character_id,
    uint32_t anim_ticks) {
  const fighter_character_animation_set_t *set;
  const fighter_animation_clip_t *clip;
```

```
  int frames_to_loop;
  int ticks_per_frame;
  int frame_index;

  if (!anim_system) {
    return NULL;
  }

  set = character_id == FIGHTER_CHARACTER_KEN ? &anim_system->ken : &anim_system->ryu;
  clip = &set->fireball_projectile;
  if (!clip->frames || clip->frame_count <= 0) {
    return NULL;
  }

  frames_to_loop = clip->frame_count < 2 ? clip->frame_count : 2;
  ticks_per_frame = clip->ticks_per_frame > 0 ? clip->ticks_per_frame : 1;
  frame_index = (int)((anim_ticks / (uint32_t)ticks_per_frame) % (uint32_t)frames_to_loop);
  return &clip->frames[frame_index];
}


static void fighter_renderer_draw_projectile_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system,
    const fighter_projectile_state_t *projectile) {
  const fighter_sprite_t *sprite;
  int hitbox_x;
  int hitbox_y;
  int hitbox_w;
  int hitbox_h;
  int draw_w;
  int draw_h;
  int draw_x;
  int draw_y;
  int flip_x;

  if (!renderer || !game || !anim_system || !projectile || !projectile->active) {
    return;
  }

  sprite = fighter_renderer_fireball_sprite(anim_system, projectile->character_id,
                                            projectile->anim_ticks);
  if (!sprite || !sprite->pixels || sprite->width <= 0 || sprite->height <= 0) {
    return;
  }

  hitbox_x =
      fighter_scale_axis(projectile->x, renderer->fb_width, game->config.screen_width);
  hitbox_y =
      fighter_scale_axis(projectile->y, renderer->fb_height, game->config.screen_height);
  hitbox_w =
      fighter_scale_size_axis(game->config.projectile_width, renderer->fb_width,
                              game->config.screen_width);
  hitbox_h =
      fighter_scale_size_axis(game->config.projectile_height, renderer->fb_height,
                              game->config.screen_height);

  draw_w = hitbox_w;
  draw_h = hitbox_h;
  draw_x = hitbox_x + (hitbox_w - draw_w) / 2;
  draw_y = hitbox_y + (hitbox_h - draw_h) / 2;

  flip_x = projectile->vx < 0;
  fighter_fb_draw_sprite(renderer, sprite, draw_x, draw_y, draw_w, draw_h, flip_x);
}


static void fighter_renderer_draw_menu_fb(fighter_renderer_t *renderer,
                                          const fighter_game_t *game) {
  fighter_fb_image_t *cached_image;
  fighter_rgb_image_t *menu_image;
  unsigned int bg_primary;
```

```
  unsigned int bg_secondary;
  unsigned int box_color;
  unsigned int text_color;
  int stripe_offset;
  int box_x;
  int box_y;
  int box_w;
  int box_h;
  int title_y;
  int prompt_y;
  int title_scale;
  int prompt_scale;
  int stripe_step;
  int stripe_width;
  int frame_index;
  int i;

  if (!renderer || !game) {
    return;
  }

  frame_index = fighter_game_menu_animation_frame(game);
  cached_image = &renderer->menu_frame_cache[frame_index & 1];
  if (!cached_image->pixels) {
    cached_image = &renderer->menu_frame_cache[0];
  }
  if (cached_image->pixels) {
    fighter_fb_draw_cached_image(renderer, cached_image);
    return;
  }

  menu_image = &renderer->menu_frames[frame_index & 1];
  if (!menu_image->pixels) {
    menu_image = &renderer->menu_frames[0];
  }
  if (menu_image->pixels) {
    fighter_fb_draw_rgb_image_fit(renderer, menu_image);
    return;
  }

  bg_primary = fighter_game_menu_animation_frame(game)
                  ? fighter_fb_color(renderer, 24, 69, 110)
                  : fighter_fb_color(renderer, 109, 31, 62);
  bg_secondary = fighter_game_menu_animation_frame(game)
                    ? fighter_fb_color(renderer, 255, 176, 59)
                    : fighter_fb_color(renderer, 52, 164, 196);
  box_color = fighter_fb_color(renderer, 18, 22, 32);
  text_color = fighter_fb_color(renderer, 248, 245, 230);
  stripe_offset = fighter_scale_axis((int)(game->frame_counter % 120U),
                                     renderer->fb_width, 640);
  box_x = fighter_scale_axis(320 - 170, renderer->fb_width, 640);
  box_y = fighter_scale_axis(90, renderer->fb_height, 480);
  box_w = fighter_scale_size_axis(340, renderer->fb_width, 640);
  box_h = fighter_scale_size_axis(140, renderer->fb_height, 480);
  title_y = fighter_scale_axis(120, renderer->fb_height, 480);
  prompt_y = fighter_scale_axis(155, renderer->fb_height, 480);
  title_scale = fighter_scale_text_size(renderer, 4);
  prompt_scale = fighter_scale_text_size(renderer, 2);
  stripe_step = fighter_scale_size_axis(120, renderer->fb_width, 640);
  stripe_width = fighter_scale_size_axis(42, renderer->fb_width, 640);

  fighter_fb_fill_rect(renderer, 0, 0, renderer->fb_width, renderer->fb_height,
                       bg_primary);
  for (i = -renderer->fb_height; i < renderer->fb_width + renderer->fb_height;
       i += stripe_step) {
    fighter_fb_fill_rect(renderer, i + stripe_offset, 0, stripe_width,
                         renderer->fb_height, bg_secondary);
  }

  fighter_fb_fill_rect(renderer, box_x, box_y, box_w, box_h, box_color);
  fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, title_y,
                                "PHASE 1 FIGHTER", title_scale, text_color);
  fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, prompt_y,
```

```
                                "PRESS ANY KEY", prompt_scale, text_color);
}


static void fighter_renderer_draw_playfield_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system,
    int draw_overlay) {
  unsigned int sky_color;
  unsigned int floor_color;
  unsigned int ui_text_color;
  unsigned int p1_bar_fg;
  unsigned int p2_bar_fg;
  unsigned int bar_bg;
  unsigned int bar_border;
  int floor_y;
  int bar_w;
  int bar_h;
  int p1_bar_x;
  int p2_bar_x;
  int bar_y;
  int timer_y;
  int label_y;
  int timer_scale;
  int label_scale;
  char timer_text[16];

  (void)draw_overlay;

  if (!renderer || !game) {
    return;
  }

  sky_color = fighter_fb_color(renderer, 120, 180, 255);
  floor_color = fighter_fb_color(renderer, 70, 120, 70);
  ui_text_color = fighter_fb_color(renderer, 248, 245, 230);
  p1_bar_fg = fighter_fb_color(renderer, 210, 60, 60);
  p2_bar_fg = fighter_fb_color(renderer, 60, 120, 220);
  bar_bg = fighter_fb_color(renderer, 40, 40, 40);
  bar_border = fighter_fb_color(renderer, 230, 230, 230);

  if (renderer->background_cache.pixels) {
    fighter_fb_draw_cached_image(renderer, &renderer->background_cache);
  } else {
    fighter_fb_clear(renderer, sky_color);
    floor_y = fighter_scale_axis(game->config.floor_y, renderer->fb_height,
                                 game->config.screen_height);
    fighter_fb_fill_rect(renderer, 0, floor_y, renderer->fb_width,
                         renderer->fb_height - floor_y, floor_color);
  }

  if (anim_system) {
    fighter_renderer_draw_player_fb(renderer, game, anim_system, 0);
    fighter_renderer_draw_player_fb(renderer, game, anim_system, 1);
    fighter_renderer_draw_projectile_fb(renderer, game, anim_system,
                                        &game->projectiles[0]);
    fighter_renderer_draw_projectile_fb(renderer, game, anim_system,
                                        &game->projectiles[1]);
  }

  bar_w = fighter_scale_size_axis(220, renderer->fb_width, 640);
  bar_h = fighter_scale_size_axis(18, renderer->fb_height, 480);
  p1_bar_x = fighter_scale_axis(20, renderer->fb_width, 640);
  p2_bar_x = fighter_scale_axis(400, renderer->fb_width, 640);
  bar_y = fighter_scale_axis(18, renderer->fb_height, 480);
  timer_y = fighter_scale_axis(18, renderer->fb_height, 480);
  label_y = fighter_scale_axis(44, renderer->fb_height, 480);
  timer_scale = fighter_scale_text_size(renderer, 3);
  label_scale = fighter_scale_text_size(renderer, 1);

  fighter_fb_draw_hp_bar(renderer, p1_bar_x, bar_y, bar_w, bar_h, game->players[0].hp,
                         game->config.max_hp, p1_bar_fg, bar_bg, bar_border);
```

```
  fighter_fb_draw_hp_bar(renderer, p2_bar_x, bar_y, bar_w, bar_h, game->players[1].hp,
                         game->config.max_hp, p2_bar_fg, bar_bg, bar_border);

  fighter_fb_draw_text(renderer, p1_bar_x, label_y, "P1", label_scale, ui_text_color);
  fighter_fb_draw_text(renderer, p2_bar_x + bar_w - 18 * label_scale, label_y, "P2",
                       label_scale, ui_text_color);

  snprintf(timer_text, sizeof(timer_text), "%02d",
           fighter_game_round_seconds_remaining(game));
  fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, timer_y, timer_text,
                                timer_scale, ui_text_color);
}


static void fighter_renderer_draw_game_over_fb(
    fighter_renderer_t *renderer,
    const fighter_game_t *game,
    const fighter_animation_system_t *anim_system) {
  const char *winner_text;
  unsigned int text_color;
  unsigned int box_color;
  int title_scale;
  int sub_scale;
  int title_y;
  int winner_y;
  int restart_y;
  int menu_y;
  int box_x;
  int box_y;
  int box_w;
  int box_h;

  fighter_renderer_draw_playfield_fb(renderer, game, anim_system, 0);
  text_color = fighter_fb_color(renderer, 248, 245, 230);
  box_color = fighter_fb_color(renderer, 0, 0, 0);
  title_scale = fighter_scale_text_size(renderer, 3);
  sub_scale = fighter_scale_text_size(renderer, 2);
  title_y = fighter_scale_axis(145, renderer->fb_height, 480);
  winner_y = fighter_scale_axis(190, renderer->fb_height, 480);
  restart_y = fighter_scale_axis(225, renderer->fb_height, 480);
  menu_y = fighter_scale_axis(250, renderer->fb_height, 480);

  box_x = fighter_scale_axis(160, renderer->fb_width, 640);
  box_y = fighter_scale_axis(120, renderer->fb_height, 480);
  box_w = fighter_scale_size_axis(320, renderer->fb_width, 640);
  box_h = fighter_scale_size_axis(160, renderer->fb_height, 480);

  fighter_fb_fill_rect(renderer, box_x, box_y, box_w, box_h, box_color);

  switch (game->winner) {
    case FIGHTER_WINNER_PLAYER1:
      winner_text = "P1 WIN";
      break;
    case FIGHTER_WINNER_PLAYER2:
      winner_text = "P2 WIN";
      break;
    case FIGHTER_WINNER_DRAW:
      winner_text = "DRAW";
      break;
    case FIGHTER_WINNER_NONE:
    default:
      winner_text = "WAIT";
      break;
  }

  fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, title_y,
                                "GAME OVER", title_scale, text_color);
  fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, winner_y,
                                winner_text, sub_scale, text_color);
  if (fighter_game_game_over_ready(game)) {
    fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, restart_y,
                                  "JK RESTART", sub_scale, text_color);
    fighter_fb_draw_centered_text(renderer, renderer->fb_width / 2, menu_y,
```

```
                                  "L TO MENU", sub_scale, text_color);
  }
}


static void fighter_renderer_draw_console(fighter_renderer_t *renderer,
                                          const fighter_game_t *game) {
  int game_changed;
  int player_changed[FIGHTER_PLAYER_COUNT];
  int should_print;
  int i;

  if (!renderer || !game) {
    return;
  }

  game_changed = !renderer->last_console_valid ||
                 renderer->last_console_state != game->state ||
                 renderer->last_console_winner != game->winner ||
                 renderer->last_console_finish_reason != game->finish_reason ||
                 renderer->last_console_ready !=
                     fighter_game_game_over_ready(game);
  should_print = game_changed;

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    player_changed[i] =
        !renderer->last_console_valid ||
        fighter_renderer_console_player_changed(&renderer->last_console_players[i],
                                                &game->players[i]) ||
        game->players[i].combat_result != FIGHTER_COMBAT_RESULT_NONE ||
        game->players[i].event_flags != FIGHTER_PLAYER_EVENT_NONE;
    if (player_changed[i]) {
      should_print = 1;
    }
  }

  if (!should_print) {
    return;
  }

  renderer->last_console_frame = game->frame_counter;
  renderer->last_console_state = game->state;
  renderer->last_console_winner = game->winner;
  renderer->last_console_finish_reason = game->finish_reason;
  renderer->last_console_ready = fighter_game_game_over_ready(game);
  renderer->last_console_valid = 1;
  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    renderer->last_console_players[i] = game->players[i];
  }

  switch (game->state) {
    case FIGHTER_GAME_STATE_MENU:
      printf("[frame %u] GAME state=%s winner=%s finish=%s\n", game->frame_counter,
             fighter_renderer_game_state_name(game->state),
             fighter_renderer_winner_name(game->winner),
             fighter_renderer_finish_reason_name(game->finish_reason));
      fighter_renderer_print_console_player("P1", &game->players[0]);
      fighter_renderer_print_console_player("P2", &game->players[1]);
      break;
    case FIGHTER_GAME_STATE_PLAYING:
      printf("[frame %u] GAME state=%s timer=%d winner=%s finish=%s\n",
             game->frame_counter, fighter_renderer_game_state_name(game->state),
             fighter_game_round_seconds_remaining(game),
             fighter_renderer_winner_name(game->winner),
             fighter_renderer_finish_reason_name(game->finish_reason));
      fighter_renderer_print_console_player("P1", &game->players[0]);
      fighter_renderer_print_console_player("P2", &game->players[1]);
      break;
    case FIGHTER_GAME_STATE_GAME_OVER:
      printf("[frame %u] GAME state=%s winner=%s finish=%s ready=%d\n",
             game->frame_counter, fighter_renderer_game_state_name(game->state),
             fighter_renderer_winner_name(game->winner),
             fighter_renderer_finish_reason_name(game->finish_reason),
```

```
            fighter_game_game_over_ready(game));
      fighter_renderer_print_console_player("P1", &game->players[0]);
      fighter_renderer_print_console_player("P2", &game->players[1]);
      break;
    default:
      break;
  }
}


int fighter_renderer_init(fighter_renderer_t *renderer,
                          const fighter_renderer_options_t *options) {
  fighter_renderer_options_t local_options;

  if (!renderer) {
    return -1;
  }

  fighter_renderer_options_init(&local_options);
  if (options) {
    local_options = *options;
  }

  memset(renderer, 0, sizeof(*renderer));
  renderer->backend = FIGHTER_RENDERER_BACKEND_CONSOLE;
  renderer->console_interval_frames = local_options.console_interval_frames;
  snprintf(renderer->init_status, sizeof(renderer->init_status),
           "console renderer active");

  renderer->fb_fd = -1;
  renderer->vga_mem_fd = -1;
  if (local_options.prefer_framebuffer && fighter_renderer_init_mmio(renderer) == 0) {
    return 0;
  }

  {
    static const char *const k_default_framebuffer_paths[] = {
        "/dev/fb0",
        "/dev/fb1",
        "/dev/graphics/fb0",
        "/dev/graphics/fb1",
    };
    const char *explicit_fb_path = local_options.framebuffer_path;
    int initialized = 0;
    int attempted_framebuffer = 0;
    int i;
  if (local_options.prefer_framebuffer) {
    for (i = 0; i < (int)(sizeof(k_default_framebuffer_paths) /
                          sizeof(k_default_framebuffer_paths[0]));
         ++i) {
      struct fb_fix_screeninfo fix_info;
      struct fb_var_screeninfo var_info;
      const char *fb_path;
      int open_errno;

      if (explicit_fb_path && explicit_fb_path[0] != '\0' && i > 0) {
        break;
      }
      fb_path = (explicit_fb_path && explicit_fb_path[0] != '\0')
                    ? explicit_fb_path
                    : k_default_framebuffer_paths[i];
      attempted_framebuffer = 1;

      renderer->fb_fd = open(fb_path, O_RDWR);
      if (renderer->fb_fd < 0) {
        open_errno = errno;
        snprintf(renderer->init_status, sizeof(renderer->init_status),
                 "framebuffer open failed on %s: %s", fb_path,
                 strerror(open_errno));
        continue;
      }

      if (ioctl(renderer->fb_fd, FBIOGET_FSCREENINFO, &fix_info) != 0 ||
```

```
          ioctl(renderer->fb_fd, FBIOGET_VSCREENINFO, &var_info) != 0) {
        open_errno = errno;
        snprintf(renderer->init_status, sizeof(renderer->init_status),
                 "framebuffer ioctl failed on %s: %s", fb_path,
                 strerror(open_errno));
        close(renderer->fb_fd);
        renderer->fb_fd = -1;
        continue;
      }

      if (var_info.bits_per_pixel != 16 && var_info.bits_per_pixel != 32) {
        snprintf(renderer->init_status, sizeof(renderer->init_status),
                 "framebuffer %s has unsupported bpp=%u", fb_path,
                 (unsigned int)var_info.bits_per_pixel);
        close(renderer->fb_fd);
        renderer->fb_fd = -1;
        continue;
      }

      renderer->fb_width = (int)var_info.xres;
      renderer->fb_height = (int)var_info.yres;
      renderer->fb_stride = (int)fix_info.line_length;
      renderer->fb_bpp = (int)var_info.bits_per_pixel;
      renderer->fb_data_length =
          (unsigned long)(renderer->fb_stride * renderer->fb_height);
      renderer->fb_data =
          mmap(NULL, renderer->fb_data_length, PROT_READ | PROT_WRITE, MAP_SHARED,
               renderer->fb_fd, 0);
      if (renderer->fb_data == MAP_FAILED) {
        open_errno = errno;
        renderer->fb_data = NULL;
        snprintf(renderer->init_status, sizeof(renderer->init_status),
                 "framebuffer mmap failed on %s: %s", fb_path,
                 strerror(open_errno));
        close(renderer->fb_fd);
        renderer->fb_fd = -1;
        continue;
      }

      renderer->fb_backbuffer = (unsigned char *)malloc(renderer->fb_data_length);
      renderer->fb_backbuffer_length = renderer->fb_data_length;
      if (renderer->fb_backbuffer) {
        memset(renderer->fb_backbuffer, 0, renderer->fb_backbuffer_length);
      } else {
        renderer->fb_backbuffer_length = 0;
      }
      renderer->backend = FIGHTER_RENDERER_BACKEND_FRAMEBUFFER;
      snprintf(renderer->framebuffer_path_used,
               sizeof(renderer->framebuffer_path_used), "%s", fb_path);
      snprintf(renderer->init_status, sizeof(renderer->init_status),
               "framebuffer active on %s (%dx%d %dbpp)", fb_path,
               renderer->fb_width, renderer->fb_height, renderer->fb_bpp);
      initialized = 1;
      break;
    }
  }

  if (initialized) {
    fighter_renderer_load_assets(renderer);
  } else if (!local_options.prefer_framebuffer) {
    snprintf(renderer->init_status, sizeof(renderer->init_status),
             "console renderer forced by option");
  } else if (attempted_framebuffer && !explicit_fb_path) {
    snprintf(renderer->init_status, sizeof(renderer->init_status),
             "no Linux framebuffer device found; tried /dev/fb0, /dev/fb1, "
             "/dev/graphics/fb0, and /dev/graphics/fb1");
  }
  }

  return 0;
}
```

```
1886 void fighter_renderer_close(fighter_renderer_t *renderer) {
1887   if (!renderer) {
1888     return;
1889   }
1890
1891   int i;
1892
1893   fighter_renderer_close_mmio(renderer);
1894
1895   for (i = 0; i < 2; ++i) {
1896     fighter_rgb_image_reset(&renderer->menu_frames[i]);
1897     fighter_fb_image_reset(&renderer->menu_frame_cache[i]);
1898   }
1899   fighter_rgb_image_reset(&renderer->background_image);
1900   fighter_fb_image_reset(&renderer->background_cache);
1901
1902   free(renderer->fb_backbuffer);
1903   renderer->fb_backbuffer = NULL;
1904   renderer->fb_backbuffer_length = 0;
1905   if (renderer->fb_data) {
1906     munmap(renderer->fb_data, renderer->fb_data_length);
1907   }
1908   if (renderer->fb_fd >= 0) {
1909     close(renderer->fb_fd);
1910   }
1911 }
1912
1913
1914 void fighter_renderer_draw(fighter_renderer_t *renderer,
1915                            const fighter_game_t *game,
1916                            const fighter_animation_system_t *anim_system) {
1917   if (!renderer || !game) {
1918     return;
1919   }
1920
1921   (void)anim_system;
1922
1923   if (renderer->backend == FIGHTER_RENDERER_BACKEND_MMIO) {
1924     fighter_renderer_draw_mmio(renderer, game, anim_system);
1925     return;
1926   }
1927
1928   if (renderer->backend == FIGHTER_RENDERER_BACKEND_FRAMEBUFFER) {
1929     switch (game->state) {
1930       case FIGHTER_GAME_STATE_MENU:
1931         fighter_renderer_draw_menu_fb(renderer, game);
1932         break;
1933       case FIGHTER_GAME_STATE_PLAYING:
1934         fighter_renderer_draw_playfield_fb(renderer, game, anim_system, 0);
1935         break;
1936       case FIGHTER_GAME_STATE_GAME_OVER:
1937         fighter_renderer_draw_game_over_fb(renderer, game, anim_system);
1938         break;
1939       default:
1940         break;
1941     }
1942     fighter_fb_present(renderer);
1943     return;
1944   }
1945
1946   fighter_renderer_draw_console(renderer, game);
1947 }
1948
1949
1950 const char *fighter_renderer_backend_name(const fighter_renderer_t *renderer) {
1951   if (!renderer) {
1952     return "unknown";
1953   }
1954
1955   switch (renderer->backend) {
1956     case FIGHTER_RENDERER_BACKEND_FRAMEBUFFER:
1957       return "framebuffer";
1958     case FIGHTER_RENDERER_BACKEND_MMIO:
```

```
      return "mmio";
    case FIGHTER_RENDERER_BACKEND_CONSOLE:
    default:
      return "console";
  }
}


const char *fighter_renderer_active_framebuffer_path(
    const fighter_renderer_t *renderer) {
  if (!renderer || renderer->framebuffer_path_used[0] == '\0') {
    return "none";
  }
  return renderer->framebuffer_path_used;
}


const char *fighter_renderer_status_detail(const fighter_renderer_t *renderer) {
  if (!renderer || renderer->init_status[0] == '\0') {
    return "no renderer status";
  }
  return renderer->init_status;
}
```

A.1.9 sw/tests/test_phase1.c

```
#include "fighter_game.h"
#include "fighter_animation.h"

#include <stdio.h>
#include <string.h>

static int g_failures = 0;

#define EXPECT_TRUE(expr)                                                      \
  do {                                                                         \
    if (!(expr)) {                                                             \
      fprintf(stderr, "EXPECT_TRUE failed at %s:%d: %s\n", __FILE__, __LINE__,   \
              #expr);                                                          \
      ++g_failures;                                                            \
    }                                                                          \
  } while (0)

#define EXPECT_EQ_INT(actual, expected)                                        \
  do {                                                                         \
    int actual_value = (actual);                                               \
    int expected_value = (expected);                                           \
    if (actual_value != expected_value) {                                      \
      fprintf(stderr,                                                           \
              "EXPECT_EQ_INT failed at %s:%d: got %d expected %d\n", __FILE__,   \
              __LINE__, actual_value, expected_value);                         \
      ++g_failures;                                                            \
    }                                                                          \
  } while (0)

static fighter_game_config_t short_config(void) {
  fighter_game_config_t config;

  fighter_game_config_default(&config);
  config.game_over_anim_frames = 3;
  config.round_duration_frames = 10;
  return config;
}

static int ground_y(const fighter_game_t *game) {
  return game->config.floor_y - game->config.player_height;
}

static void clear_inputs(fighter_player_result_t inputs[2]) {
  memset(inputs, 0, sizeof(fighter_player_result_t) * 2);
```

```
}

static void start_round(fighter_game_t *game,
                        fighter_audio_command_list_t *commands,
                        fighter_player_result_t inputs[2]) {
  clear_inputs(inputs);
  inputs[0].any_input_pressed = 1;
  fighter_game_tick(game, inputs, commands);
}

static void test_menu_transition(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];

  fighter_game_init(&game, NULL);
  clear_inputs(inputs);

  fighter_game_tick(&game, inputs, &commands);
  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_MENU);
  EXPECT_EQ_INT((int)commands.count, 1);
  EXPECT_EQ_INT(commands.commands[0].type, FIGHTER_AUDIO_COMMAND_START_LOOP);
  EXPECT_EQ_INT(commands.commands[0].track, FIGHTER_AUDIO_TRACK_MENU_BGM);

  clear_inputs(inputs);
  inputs[0].any_input_pressed = 1;
  fighter_game_tick(&game, inputs, &commands);
  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_PLAYING);
  EXPECT_EQ_INT((int)commands.count, 2);
  EXPECT_EQ_INT(commands.commands[0].type, FIGHTER_AUDIO_COMMAND_STOP_LOOP);
  EXPECT_EQ_INT(commands.commands[1].type, FIGHTER_AUDIO_COMMAND_PLAY_ONCE);
}

static void test_exit_back_to_menu(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);

  clear_inputs(inputs);
  inputs[1].exit_requested = 1;
  fighter_game_tick(&game, inputs, &commands);
  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_MENU);
  EXPECT_EQ_INT(game.finish_reason, FIGHTER_FINISH_REASON_EXIT);
  EXPECT_EQ_INT((int)commands.count, 1);
  EXPECT_EQ_INT(commands.commands[0].type, FIGHTER_AUDIO_COMMAND_START_LOOP);
}

static void test_knockout_transition(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  fighter_game_config_t config;
  int i;

  config = short_config();
  fighter_game_init(&game, &config);
  start_round(&game, &commands, inputs);

  game.players[0].x = 260;
  game.players[1].x = 300;
  game.players[1].hp = 12;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_FIREBALL;
  fighter_game_tick(&game, inputs, &commands);
  clear_inputs(inputs);
  for (i = 0; i < 20; ++i) {
    fighter_game_tick(&game, inputs, &commands);
    if (game.state == FIGHTER_GAME_STATE_GAME_OVER) {
```

```
      break;
    }
  }

  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_GAME_OVER);
  EXPECT_EQ_INT(game.winner, FIGHTER_WINNER_PLAYER1);
  EXPECT_EQ_INT(game.finish_reason, FIGHTER_FINISH_REASON_KO);
  EXPECT_EQ_INT((int)commands.count, 1);
  EXPECT_EQ_INT(commands.commands[0].track, FIGHTER_AUDIO_TRACK_GAME_OVER);
}

static void test_game_over_restart_gate(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  fighter_game_config_t config;

  config = short_config();
  fighter_game_init(&game, &config);
  game.state = FIGHTER_GAME_STATE_GAME_OVER;
  game.state_frames = 1;
  game.winner = FIGHTER_WINNER_PLAYER2;
  game.menu_bgm_active = 0;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  fighter_game_tick(&game, inputs, &commands);
  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_GAME_OVER);

  game.state_frames = (unsigned int)config.game_over_anim_frames;
  clear_inputs(inputs);
  inputs[1].guard_pressed = 1;
  fighter_game_tick(&game, inputs, &commands);
  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_PLAYING);
  EXPECT_EQ_INT((int)commands.count, 1);
  EXPECT_EQ_INT(commands.commands[0].type, FIGHTER_AUDIO_COMMAND_PLAY_ONCE);
}

static void test_timeout_transition(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  fighter_game_config_t config;

  fighter_game_config_default(&config);
  config.round_duration_frames = 1;
  config.game_over_anim_frames = 2;
  fighter_game_init(&game, &config);

  start_round(&game, &commands, inputs);

  game.players[0].hp = 40;
  game.players[1].hp = 40;
  clear_inputs(inputs);
  fighter_game_tick(&game, inputs, &commands);

  EXPECT_EQ_INT(game.state, FIGHTER_GAME_STATE_GAME_OVER);
  EXPECT_EQ_INT(game.winner, FIGHTER_WINNER_DRAW);
  EXPECT_EQ_INT(game.finish_reason, FIGHTER_FINISH_REASON_TIME_OUT);
}

static void test_hit_confirm_state_machine(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int i;

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);

  game.players[0].x = 260;
  game.players[1].x = 300;

```

```
  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_NORMAL;
  fighter_game_tick(&game, inputs, &commands);

  clear_inputs(inputs);
  for (i = 0; i < 8; ++i) {
    fighter_game_tick(&game, inputs, &commands);
    if (game.players[0].attack_phase == FIGHTER_ATTACK_PHASE_HIT_CONFIRM) {
      break;
    }
  }

  EXPECT_EQ_INT(game.players[0].attack_phase,
                FIGHTER_ATTACK_PHASE_HIT_CONFIRM);
  EXPECT_EQ_INT(game.players[0].combat_result, FIGHTER_COMBAT_RESULT_HIT);
  EXPECT_TRUE((game.players[0].event_flags & FIGHTER_PLAYER_EVENT_HIT) != 0);
  EXPECT_EQ_INT(game.players[1].visual_state, FIGHTER_VISUAL_STATE_HIT);
  EXPECT_EQ_INT(game.players[1].combat_result, FIGHTER_COMBAT_RESULT_HIT);
  EXPECT_TRUE(game.players[1].hurt_visual_frames > 0);
}

static void test_block_stun_fields(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int i;

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);

  game.players[0].x = 260;
  game.players[1].x = 300;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_NORMAL;
  inputs[1].guard_held = 1;
  fighter_game_tick(&game, inputs, &commands);

  for (i = 0; i < 8; ++i) {
    clear_inputs(inputs);
    inputs[1].guard_held = 1;
    fighter_game_tick(&game, inputs, &commands);
    if (game.players[1].block_stun_frames > 0) {
      break;
    }
  }

  EXPECT_TRUE(game.players[1].block_stun_frames > 0);
  EXPECT_EQ_INT(game.players[1].visual_state,
                FIGHTER_VISUAL_STATE_BLOCK_STUN);
  EXPECT_EQ_INT(game.players[0].attack_phase,
                FIGHTER_ATTACK_PHASE_BLOCK_CONFIRM);
  EXPECT_EQ_INT(game.players[0].combat_result,
                FIGHTER_COMBAT_RESULT_BLOCKED);
  EXPECT_EQ_INT(game.players[1].combat_result,
                FIGHTER_COMBAT_RESULT_BLOCKED);
  EXPECT_TRUE((game.players[1].event_flags & FIGHTER_PLAYER_EVENT_BLOCK) != 0);
}

static void test_crouch_guard_state(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);

  clear_inputs(inputs);
  inputs[0].crouch_held = 1;
  inputs[0].guard_held = 1;
  fighter_game_tick(&game, inputs, &commands);
```

```
264
265   EXPECT_EQ_INT(game.players[0].visual_state,
266                 FIGHTER_VISUAL_STATE_CROUCH_GUARD);
267 }
268
269 static void test_grounded_jump_attack_requires_airborne(void) {
270   fighter_game_t game;
271   fighter_audio_command_list_t commands;
272   fighter_player_result_t inputs[2];
273
274   fighter_game_init(&game, NULL);
275   start_round(&game, &commands, inputs);
276
277   clear_inputs(inputs);
278   inputs[0].jump_pressed = 1;
279   inputs[0].jump_held = 1;
280   inputs[0].attack_pressed = 1;
281   inputs[0].attack_command = FIGHTER_ATTACK_JUMP_ATTACK;
282   fighter_game_tick(&game, inputs, &commands);
283
284   EXPECT_TRUE(game.players[0].vy < 0);
285   EXPECT_EQ_INT(game.players[0].attack_phase, FIGHTER_ATTACK_PHASE_NONE);
286 }
287
288 static void test_directional_jump_moves_horizontally(void) {
289   fighter_game_t game;
290   fighter_audio_command_list_t commands;
291   fighter_player_result_t inputs[2];
292   int start_x;
293
294   fighter_game_init(&game, NULL);
295   start_round(&game, &commands, inputs);
296
297   start_x = game.players[0].x;
298   clear_inputs(inputs);
299   inputs[0].jump_pressed = 1;
300   inputs[0].jump_held = 1;
301   inputs[0].move_right = 1;
302   fighter_game_tick(&game, inputs, &commands);
303
304   EXPECT_TRUE(game.players[0].vy < 0);
305   EXPECT_TRUE(game.players[0].vx > 0);
306
307   clear_inputs(inputs);
308   fighter_game_tick(&game, inputs, &commands);
309   EXPECT_TRUE(game.players[0].x > start_x);
310   EXPECT_TRUE(game.players[0].y < ground_y(&game));
311 }
312
313 static void test_airborne_movement_and_attack_restrictions(void) {
314   fighter_game_t game;
315   fighter_audio_command_list_t commands;
316   fighter_player_result_t inputs[2];
317   int initial_x;
318
319   fighter_game_init(&game, NULL);
320   start_round(&game, &commands, inputs);
321
322   game.players[0].x = 220;
323   game.players[0].y = ground_y(&game) - 28;
324   game.players[0].vy = -3;
325   game.players[1].x = 360;
326   initial_x = game.players[0].x;
327
328   clear_inputs(inputs);
329   inputs[0].move_right = 1;
330   inputs[0].attack_pressed = 1;
331   inputs[0].attack_command = FIGHTER_ATTACK_DRAGON_PUNCH;
332   fighter_game_tick(&game, inputs, &commands);
333
334   EXPECT_EQ_INT(game.players[0].x, initial_x);
335   EXPECT_EQ_INT(game.players[0].attack_phase, FIGHTER_ATTACK_PHASE_NONE);
336   EXPECT_TRUE(game.players[0].y < ground_y(&game));
```

```
337
338   clear_inputs(inputs);
339   inputs[0].jump_held = 1;
340   inputs[0].attack_pressed = 1;
341   inputs[0].attack_command = FIGHTER_ATTACK_JUMP_ATTACK;
342   fighter_game_tick(&game, inputs, &commands);
343
344   EXPECT_EQ_INT(game.players[0].attack_phase, FIGHTER_ATTACK_PHASE_STARTUP);
345 }
346
347 static void test_grounded_overlap_still_resolves(void) {
348   fighter_game_t game;
349   fighter_audio_command_list_t commands;
350   fighter_player_result_t inputs[2];
351
352   fighter_game_init(&game, NULL);
353   start_round(&game, &commands, inputs);
354
355   game.players[0].x = 280;
356   game.players[1].x = 300;
357   game.players[0].y = ground_y(&game);
358   game.players[1].y = ground_y(&game);
359
360   clear_inputs(inputs);
361   fighter_game_tick(&game, inputs, &commands);
362
363   EXPECT_TRUE(game.players[0].x + game.config.player_width <= game.players[1].x);
364   EXPECT_EQ_INT(game.players[0].facing, 1);
365   EXPECT_EQ_INT(game.players[1].facing, -1);
366 }
367
368 static void test_dragon_punch_lifts_player(void) {
369   fighter_game_t game;
370   fighter_audio_command_list_t commands;
371   fighter_player_result_t inputs[2];
372   int initial_y;
373   int i;
374
375   fighter_game_init(&game, NULL);
376   start_round(&game, &commands, inputs);
377   initial_y = game.players[0].y;
378
379   clear_inputs(inputs);
380   inputs[0].attack_pressed = 1;
381   inputs[0].attack_command = FIGHTER_ATTACK_DRAGON_PUNCH;
382   fighter_game_tick(&game, inputs, &commands);
383
384   clear_inputs(inputs);
385   for (i = 0; i < 8; ++i) {
386     fighter_game_tick(&game, inputs, &commands);
387     if (game.players[0].y < initial_y) {
388       break;
389     }
390   }
391
392   EXPECT_TRUE(game.players[0].y < initial_y);
393 }
394
395 static void test_attack_locks_out_movement(void) {
396   fighter_game_t game;
397   fighter_audio_command_list_t commands;
398   fighter_player_result_t inputs[2];
399   int attack_x;
400
401   fighter_game_init(&game, NULL);
402   start_round(&game, &commands, inputs);
403
404   clear_inputs(inputs);
405   inputs[0].attack_pressed = 1;
406   inputs[0].attack_command = FIGHTER_ATTACK_NORMAL;
407   fighter_game_tick(&game, inputs, &commands);
408   attack_x = game.players[0].x;
409
```

```
  clear_inputs(inputs);
  inputs[0].move_right = 1;
  fighter_game_tick(&game, inputs, &commands);

  EXPECT_EQ_INT(game.players[0].attack_phase, FIGHTER_ATTACK_PHASE_STARTUP);
  EXPECT_EQ_INT(game.players[0].x, attack_x);
}

static void test_fireball_animation_not_cut_short(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int i;

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);
  game.players[1].x = 520;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_FIREBALL;
  fighter_game_tick(&game, inputs, &commands);

  for (i = 0; i < 20; ++i) {
    clear_inputs(inputs);
    inputs[0].move_right = 1;
    fighter_game_tick(&game, inputs, &commands);
  }

  EXPECT_TRUE(game.players[0].attack_phase != FIGHTER_ATTACK_PHASE_NONE);
  EXPECT_EQ_INT(game.players[0].last_attack, FIGHTER_ATTACK_FIREBALL);
}

static void test_fireball_projectile_hits_and_disappears(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int saw_projectile;
  int i;

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);
  game.players[0].x = 120;
  game.players[1].x = 360;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_FIREBALL;
  fighter_game_tick(&game, inputs, &commands);

  saw_projectile = 0;
  for (i = 0; i < 60; ++i) {
    clear_inputs(inputs);
    fighter_game_tick(&game, inputs, &commands);
    if (game.projectiles[0].active) {
      saw_projectile = 1;
    }
    if (game.players[1].hurt_visual_frames > 0) {
      break;
    }
  }

  EXPECT_TRUE(saw_projectile);
  EXPECT_TRUE(game.players[1].hp < game.config.max_hp);
  EXPECT_EQ_INT(game.projectiles[0].active, 0);
}

static void test_fireball_projectiles_cancel_each_other(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int saw_two_projectiles;
  int i;
```

```
483
484   fighter_game_init(&game, NULL);
485   start_round(&game, &commands, inputs);
486   game.players[0].x = 120;
487   game.players[1].x = 420;
488
489   clear_inputs(inputs);
490   inputs[0].attack_pressed = 1;
491   inputs[0].attack_command = FIGHTER_ATTACK_FIREBALL;
492   inputs[1].attack_pressed = 1;
493   inputs[1].attack_command = FIGHTER_ATTACK_FIREBALL;
494   fighter_game_tick(&game, inputs, &commands);
495
496   saw_two_projectiles = 0;
497   for (i = 0; i < 80; ++i) {
498     clear_inputs(inputs);
499     fighter_game_tick(&game, inputs, &commands);
500     if (game.projectiles[0].active && game.projectiles[1].active) {
501       saw_two_projectiles = 1;
502     }
503     if (saw_two_projectiles &&
504         !game.projectiles[0].active && !game.projectiles[1].active) {
505       break;
506     }
507   }
508
509   EXPECT_TRUE(saw_two_projectiles);
510   EXPECT_EQ_INT(game.projectiles[0].active, 0);
511   EXPECT_EQ_INT(game.projectiles[1].active, 0);
512   EXPECT_EQ_INT(game.players[0].hp, game.config.max_hp);
513   EXPECT_EQ_INT(game.players[1].hp, game.config.max_hp);
514 }
515
516 static void test_fireball_projectile_disappears_at_boundary(void) {
517   fighter_game_t game;
518   fighter_audio_command_list_t commands;
519   fighter_player_result_t inputs[2];
520   int i;
521
522   fighter_game_init(&game, NULL);
523   start_round(&game, &commands, inputs);
524   game.players[0].x = game.config.screen_width - game.config.player_width - 90;
525   game.players[1].x = game.config.screen_width - game.config.player_width;
526   game.players[1].y = 0;
527   game.players[1].vy = -1;
528
529   clear_inputs(inputs);
530   inputs[0].attack_pressed = 1;
531   inputs[0].attack_command = FIGHTER_ATTACK_FIREBALL;
532   fighter_game_tick(&game, inputs, &commands);
533
534   for (i = 0; i < 80; ++i) {
535     clear_inputs(inputs);
536     fighter_game_tick(&game, inputs, &commands);
537     if (!game.projectiles[0].active) {
538       break;
539     }
540   }
541
542   EXPECT_EQ_INT(game.projectiles[0].active, 0);
543   EXPECT_EQ_INT(game.players[1].hp, game.config.max_hp);
544 }
545
546 static void test_dragon_punch_animation_not_interrupted_by_input(void) {
547   fighter_game_t game;
548   fighter_audio_command_list_t commands;
549   fighter_player_result_t inputs[2];
550   int i;
551
552   fighter_game_init(&game, NULL);
553   start_round(&game, &commands, inputs);
554   game.players[1].x = 520;
555
```

```
  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_DRAGON_PUNCH;
  fighter_game_tick(&game, inputs, &commands);

  for (i = 0; i < 20; ++i) {
    clear_inputs(inputs);
    inputs[0].move_left = 1;
    inputs[0].crouch_held = 1;
    inputs[0].guard_held = 1;
    fighter_game_tick(&game, inputs, &commands);
  }

  EXPECT_TRUE(game.players[0].attack_phase != FIGHTER_ATTACK_PHASE_NONE);
  EXPECT_EQ_INT(game.players[0].last_attack, FIGHTER_ATTACK_DRAGON_PUNCH);
}

static void test_defaults_reduce_mobility(void) {
  fighter_game_config_t config;

  fighter_game_config_default(&config);
  EXPECT_EQ_INT(config.walk_speed, 3);
  EXPECT_EQ_INT(config.jump_velocity, -14);
  EXPECT_EQ_INT(config.dragon_punch_lift_velocity, -9);
  EXPECT_EQ_INT(config.projectile_speed, 6);
}

static void test_attack_hits_only_once_per_attack(void) {
  fighter_game_t game;
  fighter_audio_command_list_t commands;
  fighter_player_result_t inputs[2];
  int i;

  fighter_game_init(&game, NULL);
  start_round(&game, &commands, inputs);

  game.players[0].x = 260;
  game.players[1].x = 300;

  clear_inputs(inputs);
  inputs[0].attack_pressed = 1;
  inputs[0].attack_command = FIGHTER_ATTACK_NORMAL;
  fighter_game_tick(&game, inputs, &commands);

  clear_inputs(inputs);
  for (i = 0; i < 10; ++i) {
    fighter_game_tick(&game, inputs, &commands);
  }

  EXPECT_EQ_INT(game.players[1].hp, game.config.max_hp - 12);
}

static void test_non_looping_hold_animations_stop_on_last_frame(void) {
  fighter_animation_system_t anim_system;
  fighter_game_t game;
  int init_rc;
  int i;

  fighter_game_init(&game, NULL);
  init_rc = fighter_animation_system_init_with_roots(
      &anim_system,
      "../game_assets/sprites/RyuPPM",
      "../game_assets/sprites/KenPPM");
  EXPECT_EQ_INT(init_rc, 0);
  if (init_rc != 0) {
    return;
  }

  game.players[0].visual_state = FIGHTER_VISUAL_STATE_CROUCH;
  for (i = 0; i < 200; ++i) {
    fighter_animation_system_update(&anim_system, &game);
  }
  EXPECT_TRUE(anim_system.ryu.crouch.frame_count > 0);
```

```
  EXPECT_EQ_INT(fighter_animation_current_frame_index(&anim_system, 0),
                anim_system.ryu.crouch.frame_count - 1);

  game.players[0].visual_state = FIGHTER_VISUAL_STATE_VICTORY;
  for (i = 0; i < 200; ++i) {
    fighter_animation_system_update(&anim_system, &game);
  }
  EXPECT_TRUE(anim_system.ryu.victory.frame_count > 0);
  EXPECT_EQ_INT(fighter_animation_current_frame_index(&anim_system, 0),
                anim_system.ryu.victory.frame_count - 1);

  fighter_animation_system_close(&anim_system);
}

int main(void) {
  test_menu_transition();
  test_exit_back_to_menu();
  test_knockout_transition();
  test_game_over_restart_gate();
  test_timeout_transition();
  test_hit_confirm_state_machine();
  test_block_stun_fields();
  test_crouch_guard_state();
  test_grounded_jump_attack_requires_airborne();
  test_directional_jump_moves_horizontally();
  test_airborne_movement_and_attack_restrictions();
  test_grounded_overlap_still_resolves();
  test_dragon_punch_lifts_player();
  test_attack_locks_out_movement();
  test_fireball_animation_not_cut_short();
  test_fireball_projectile_hits_and_disappears();
  test_fireball_projectiles_cancel_each_other();
  test_fireball_projectile_disappears_at_boundary();
  test_dragon_punch_animation_not_interrupted_by_input();
  test_defaults_reduce_mobility();
  test_attack_hits_only_once_per_attack();
  test_non_looping_hold_animations_stop_on_last_frame();

  if (g_failures != 0) {
    fprintf(stderr, "phase1 tests failed: %d\n", g_failures);
    return 1;
  }

  printf("phase1 tests passed\n");
  return 0;
}
```

A.1.10 sw/fighter_animation.c

```
#include "fighter_animation.h"

#include <dirent.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FIGHTER_DEFAULT_RYU_ROOT "/root/game_assets/sprites/RyuPPM"
#define FIGHTER_DEFAULT_KEN_ROOT "/root/game_assets/sprites/KenPPM"
#define FIGHTER_REPO_RYU_ROOT "../game_assets/sprites/RyuPPM"
#define FIGHTER_REPO_KEN_ROOT "../game_assets/sprites/KenPPM"

typedef struct {
  char **items;
  int count;
} fighter_path_list_t;


static void fighter_free_sprite(fighter_sprite_t *sprite) {
  if (!sprite) {
```

```
    return;
  }
  free(sprite->pixels);
  free(sprite->source_path);
  sprite->pixels = NULL;
  sprite->source_path = NULL;
  sprite->width = 0;
  sprite->height = 0;
}


static void fighter_free_clip(fighter_animation_clip_t *clip) {
  int i;

  if (!clip) {
    return;
  }

  for (i = 0; i < clip->frame_count; ++i) {
    fighter_free_sprite(&clip->frames[i]);
  }
  free(clip->frames);
  clip->frames = NULL;
  clip->frame_count = 0;
  clip->ticks_per_frame = 0;
  clip->loop = 0;
}


static void fighter_free_animation_set(fighter_character_animation_set_t *set) {
  if (!set) {
    return;
  }

  fighter_free_clip(&set->idle);
  fighter_free_clip(&set->walk);
  fighter_free_clip(&set->crouch);
  fighter_free_clip(&set->crouch_guard);
  fighter_free_clip(&set->jump);
  fighter_free_clip(&set->guard);
  fighter_free_clip(&set->block_stun);
  fighter_free_clip(&set->hit);
  fighter_free_clip(&set->ko);
  fighter_free_clip(&set->victory);

  fighter_free_clip(&set->normal_attack);
  fighter_free_clip(&set->fireball_attack);
  fighter_free_clip(&set->fireball_projectile);
  fighter_free_clip(&set->dragon_punch_attack);
  fighter_free_clip(&set->jump_attack);
  fighter_free_clip(&set->forward_jump_attack);
  fighter_free_clip(&set->back_jump_attack);
  fighter_free_clip(&set->sweep_attack);
}


static char *fighter_strdup_local(const char *s) {
  size_t n;
  char *out;

  if (!s) {
    return NULL;
  }

  n = strlen(s);
  out = (char *)malloc(n + 1);
  if (!out) {
    return NULL;
  }
  memcpy(out, s, n + 1);
  return out;
}

```

```
95
96  static int fighter_has_rgb565_extension(const char *name) {
97    size_t len;
98
99    if (!name) {
100     return 0;
101   }
102
103   len = strlen(name);
104   if (len < 7) {
105     return 0;
106   }
107   return strcmp(name + len - 7, ".rgb565") == 0;
108 }
109
110
111 static int fighter_compare_paths(const void *lhs, const void *rhs) {
112   const char *const *a = (const char *const *)lhs;
113   const char *const *b = (const char *const *)rhs;
114   return strcmp(*a, *b);
115 }
116
117
118 static void fighter_path_list_free(fighter_path_list_t *list) {
119   int i;
120
121   if (!list) {
122     return;
123   }
124
125   for (i = 0; i < list->count; ++i) {
126     free(list->items[i]);
127   }
128   free(list->items);
129   list->items = NULL;
130   list->count = 0;
131 }
132
133
134 static int fighter_collect_rgb565_files(const char *dir_path,
135                                         fighter_path_list_t *out_list) {
136   DIR *dir;
137   struct dirent *entry;
138   fighter_path_list_t list;
139   int capacity;
140
141   if (!dir_path || !out_list) {
142     return -1;
143   }
144
145   memset(&list, 0, sizeof(list));
146   capacity = 0;
147
148   dir = opendir(dir_path);
149   if (!dir) {
150     return -1;
151   }
152
153   while ((entry = readdir(dir)) != NULL) {
154     char full_path[PATH_MAX];
155     char *stored_path;
156
157     if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
158       continue;
159     }
160     if (!fighter_has_rgb565_extension(entry->d_name)) {
161       continue;
162     }
163
164     if (snprintf(full_path, sizeof(full_path), "%s/%s", dir_path, entry->d_name) >=
165         (int)sizeof(full_path)) {
166       closedir(dir);
167       fighter_path_list_free(&list);
```

```
      return -1;
    }

    if (list.count == capacity) {
      int new_capacity = capacity == 0 ? 8 : capacity * 2;
      char **new_items =
          (char **)realloc(list.items, (size_t)new_capacity * sizeof(char *));
      if (!new_items) {
        closedir(dir);
        fighter_path_list_free(&list);
        return -1;
      }
      list.items = new_items;
      capacity = new_capacity;
    }

    stored_path = fighter_strdup_local(full_path);
    if (!stored_path) {
      closedir(dir);
      fighter_path_list_free(&list);
      return -1;
    }

    list.items[list.count++] = stored_path;
  }

  closedir(dir);

  if (list.count > 1) {
    qsort(list.items, (size_t)list.count, sizeof(char *), fighter_compare_paths);
  }

  *out_list = list;
  return 0;
}


static unsigned int fighter_read_le16(const unsigned char *data) {
  return (unsigned int)data[0] | ((unsigned int)data[1] << 8);
}


static unsigned long fighter_read_le32(const unsigned char *data) {
  return (unsigned long)data[0] | ((unsigned long)data[1] << 8) |
         ((unsigned long)data[2] << 16) | ((unsigned long)data[3] << 24);
}


static int fighter_load_rgb565(const char *path, fighter_sprite_t *out_sprite) {
  FILE *fp;
  unsigned char header[16];
  int width;
  int height;
  unsigned long data_size;
  size_t bytes_needed;
  unsigned char *pixels;
  fighter_sprite_t sprite;

  if (!path || !out_sprite) {
    return -1;
  }

  memset(&sprite, 0, sizeof(sprite));

  fp = fopen(path, "rb");
  if (!fp) {
    return -1;
  }

  if (fread(header, 1, sizeof(header), fp) != sizeof(header)) {
    fclose(fp);
    return -1;
  }
```

```

  if (memcmp(header, "R565", 4) != 0 || fighter_read_le16(header + 4) != 16 ||
      fighter_read_le16(header + 10) != 1) {
    fclose(fp);
    return -1;
  }

  width = (int)fighter_read_le16(header + 6);
  height = (int)fighter_read_le16(header + 8);
  data_size = fighter_read_le32(header + 12);
  if (width <= 0 || height <= 0) {
    fclose(fp);
    return -1;
  }

  bytes_needed = (size_t)width * (size_t)height * 2U;
  if (data_size != (unsigned long)bytes_needed) {
    fclose(fp);
    return -1;
  }

  pixels = (unsigned char *)malloc(bytes_needed);
  if (!pixels) {
    fclose(fp);
    return -1;
  }

  if (fread(pixels, 1, bytes_needed, fp) != bytes_needed) {
    free(pixels);
    fclose(fp);
    return -1;
  }

  fclose(fp);

  sprite.width = width;
  sprite.height = height;
  sprite.pixels = pixels;
  sprite.source_path = fighter_strdup_local(path);
  if (!sprite.source_path) {
    free(pixels);
    return -1;
  }

  *out_sprite = sprite;
  return 0;
}


static int fighter_load_clip_from_directory(const char *dir_path,
                                            int ticks_per_frame,
                                            int loop,
                                            fighter_animation_clip_t *out_clip) {
  fighter_path_list_t paths;
  fighter_animation_clip_t clip;
  int i;

  if (!dir_path || !out_clip) {
    return -1;
  }

  memset(&paths, 0, sizeof(paths));
  memset(&clip, 0, sizeof(clip));

  if (fighter_collect_rgb565_files(dir_path, &paths) != 0 || paths.count <= 0) {
    fighter_path_list_free(&paths);
    return -1;
  }

  clip.frames = (fighter_sprite_t *)calloc((size_t)paths.count,
                                           sizeof(fighter_sprite_t));
  if (!clip.frames) {
    fighter_path_list_free(&paths);
```

```
    return -1;
  }

  clip.frame_count = paths.count;
  clip.ticks_per_frame = ticks_per_frame > 0 ? ticks_per_frame : 1;
  clip.loop = loop ? 1 : 0;

  for (i = 0; i < paths.count; ++i) {
    if (fighter_load_rgb565(paths.items[i], &clip.frames[i]) != 0) {
      fighter_free_clip(&clip);
      fighter_path_list_free(&paths);
      return -1;
    }
  }

  fighter_path_list_free(&paths);

  *out_clip = clip;
  return 0;
}


static int fighter_join_path(char *out_path,
                             size_t out_size,
                             const char *base,
                             const char *suffix) {
  if (!out_path || out_size == 0 || !base || !suffix) {
    return -1;
  }

  if (snprintf(out_path, out_size, "%s/%s", base, suffix) >= (int)out_size) {
    return -1;
  }
  return 0;
}


static int fighter_directory_exists(const char *path) {
  DIR *dir;

  if (!path) {
    return 0;
  }

  dir = opendir(path);
  if (!dir) {
    return 0;
  }

  closedir(dir);
  return 1;
}


static int fighter_try_load_clip(const char *base_root,
                                 const char *relative_dir,
                                 int ticks_per_frame,
                                 int loop,
                                 fighter_animation_clip_t *out_clip) {
  char full_path[PATH_MAX];
    int rc;

  if (fighter_join_path(full_path, sizeof(full_path), base_root, relative_dir) != 0) {
    return -1;
  }

  rc = fighter_load_clip_from_directory(full_path, ticks_per_frame, loop, out_clip);
  if (rc != 0) {
    fprintf(stderr, "failed to load clip: %s\n", full_path);
  }

  return rc;
}
```

```
387
388
389 static int fighter_load_character_animation_set(
390     const char *base_root,
391     fighter_character_animation_set_t *set) {
392   if (!base_root || !set) {
393     return -1;
394   }
395
396   memset(set, 0, sizeof(*set));
397
398   if (fighter_try_load_clip(base_root, "idle", 10, 1, &set->idle) != 0) return -1;
399   if (fighter_try_load_clip(base_root, "walk", 6, 1, &set->walk) != 0) return -1;
400   if (fighter_try_load_clip(base_root, "crouch", 10, 0, &set->crouch) != 0) return -1;
401   if (fighter_try_load_clip(base_root, "crouch_guard", 6, 0, &set->crouch_guard) != 0) {
402     if (fighter_try_load_clip(base_root, "crouch_hit", 6, 0, &set->crouch_guard) != 0) {
403       if (fighter_try_load_clip(base_root, "crouch", 10, 0, &set->crouch_guard) != 0) {
404         fighter_free_animation_set(set);
405         return -1;
406       }
407     }
408   }
409   if (fighter_try_load_clip(base_root, "jump", 8, 1, &set->jump) != 0) return -1;
410   if (fighter_try_load_clip(base_root, "guard", 8, 1, &set->guard) != 0) return -1;
411
412   if (fighter_try_load_clip(base_root, "guard", 6, 0, &set->block_stun) != 0) {
413     fighter_free_animation_set(set);
414     return -1;
415   }
416
417   if (fighter_try_load_clip(base_root, "hit", 6, 0, &set->hit) != 0) return -1;
418   if (fighter_try_load_clip(base_root, "ko", 5, 0, &set->ko) != 0) return -1;
419
420   if (fighter_try_load_clip(base_root, "attack_normal", 4, 0,
421                             &set->normal_attack) != 0) return -1;
422   if (fighter_try_load_clip(base_root, "attack_fireball", 8, 0,
423                             &set->fireball_attack) != 0) return -1;
424   if (fighter_try_load_clip(base_root, "fireball", 4, 1,
425                             &set->fireball_projectile) != 0) {
426     if (fighter_try_load_clip(base_root, "attack_fireball", 4, 1,
427                               &set->fireball_projectile) != 0) {
428       fighter_free_animation_set(set);
429       return -1;
430     }
431   }
432   if (fighter_try_load_clip(base_root, "attack_dragon_punch", 6, 0,
433                             &set->dragon_punch_attack) != 0) return -1;
434   if (fighter_try_load_clip(base_root, "attack_jump", 5, 0,
435                             &set->jump_attack) != 0) return -1;
436
437   if (fighter_try_load_clip(base_root, "forward_jump", 8, 1,
438                             &set->forward_jump_attack) != 0) {
439     if (fighter_try_load_clip(base_root, "attack_jump", 5, 0,
440                               &set->forward_jump_attack) != 0) {
441       fighter_free_animation_set(set);
442       return -1;
443     }
444   }
445
446   if (fighter_try_load_clip(base_root, "jump", 8, 1,
447                             &set->back_jump_attack) != 0) {
448     if (fighter_try_load_clip(base_root, "attack_jump", 5, 0,
449                               &set->back_jump_attack) != 0) {
450       fighter_free_animation_set(set);
451       return -1;
452     }
453   }
454
455   if (fighter_try_load_clip(base_root, "attack_crouch", 5, 0,
456                             &set->sweep_attack) != 0) {
457     if (fighter_try_load_clip(base_root, "crouch_hit", 5, 0,
458                               &set->sweep_attack) != 0) {
459       fighter_free_animation_set(set);
```

```
      return -1;
    }
  }
  if (fighter_try_load_clip(base_root, "extras/victory_1", 6, 0, &set->victory) != 0) {
    if (fighter_try_load_clip(base_root, "extras/victory_2", 6, 0, &set->victory) != 0) {
      if (fighter_try_load_clip(base_root, "idle", 10, 0, &set->victory) != 0) {
        return -1;
      }
    }
  }

  return 0;
}

static const fighter_character_animation_set_t *
fighter_select_character_set(const fighter_animation_system_t *system,
                             fighter_character_id_t character_id) {
  if (!system) {
    return NULL;
  }

  switch (character_id) {
    case FIGHTER_CHARACTER_KEN:
      return &system->ken;
    case FIGHTER_CHARACTER_RYU:
    default:
      return &system->ryu;
  }
}

static const fighter_animation_clip_t *
fighter_select_clip_for_player(const fighter_player_state_t *player,
                               const fighter_character_animation_set_t *set) {
  if (!player || !set) {
    return NULL;
  }

  switch (player->visual_state) {
    case FIGHTER_VISUAL_STATE_IDLE:
      return &set->idle;
    case FIGHTER_VISUAL_STATE_WALK:
      return &set->walk;
    case FIGHTER_VISUAL_STATE_CROUCH:
      return &set->crouch;
    case FIGHTER_VISUAL_STATE_JUMP:
      if (player->vx == 0) {
        return &set->jump;
      }
      if (player->vx * player->facing > 0) {
        return &set->forward_jump_attack;
      }
      return &set->back_jump_attack;
    case FIGHTER_VISUAL_STATE_GUARD:
      return &set->guard;
    case FIGHTER_VISUAL_STATE_CROUCH_GUARD:
      return &set->crouch_guard;
    case FIGHTER_VISUAL_STATE_BLOCK_STUN:
      return &set->block_stun;
    case FIGHTER_VISUAL_STATE_HIT:
      return &set->hit;
    case FIGHTER_VISUAL_STATE_KO:
      return &set->ko;
    case FIGHTER_VISUAL_STATE_ATTACK:
      switch (player->last_attack) {
        case FIGHTER_ATTACK_FIREBALL:
          return &set->fireball_attack;
        case FIGHTER_ATTACK_DRAGON_PUNCH:
          return &set->dragon_punch_attack;
        case FIGHTER_ATTACK_JUMP_ATTACK:
          return &set->jump_attack;
        case FIGHTER_ATTACK_FORWARD_JUMP_ATTACK:
          return &set->forward_jump_attack;
        case FIGHTER_ATTACK_BACK_JUMP_ATTACK:
```

```
          return &set->back_jump_attack;
        case FIGHTER_ATTACK_SWEEP:
          return &set->sweep_attack;
        case FIGHTER_ATTACK_NORMAL:
        case FIGHTER_ATTACK_NONE:
        default:
          return &set->normal_attack;
      }
    case FIGHTER_VISUAL_STATE_VICTORY:
      return &set->victory;
    default:
      return &set->idle;
  }
}


static void fighter_animation_state_reset(fighter_player_animation_state_t *state,
                                          const fighter_animation_clip_t *clip) {
  if (!state) {
    return;
  }

  state->current_clip = clip;
  state->frame_index = 0;
  state->tick_in_frame = 0;
}


static void fighter_animation_state_advance(
    fighter_player_animation_state_t *state) {
  const fighter_animation_clip_t *clip;

  if (!state) {
    return;
  }

  clip = state->current_clip;
  if (!clip || clip->frame_count <= 0) {
    return;
  }

  state->tick_in_frame++;
  if (state->tick_in_frame < clip->ticks_per_frame) {
    return;
  }

  state->tick_in_frame = 0;
  state->frame_index++;

  if (state->frame_index >= clip->frame_count) {
    if (clip->loop) {
      state->frame_index = 0;
    } else {
      state->frame_index = clip->frame_count - 1;
    }
  }
}


int fighter_animation_system_init(fighter_animation_system_t *system) {
  const char *asset_root = getenv("FIGHTER_ASSET_ROOT");
  char ryu_root[PATH_MAX];
  char ken_root[PATH_MAX];

  if (asset_root && asset_root[0] != '\0') {
    if (snprintf(ryu_root, sizeof(ryu_root), "%s/sprites/RyuPPM", asset_root) >=
            (int)sizeof(ryu_root) ||
        snprintf(ken_root, sizeof(ken_root), "%s/sprites/KenPPM", asset_root) >=
            (int)sizeof(ken_root)) {
      return -1;
    }
    return fighter_animation_system_init_with_roots(system, ryu_root, ken_root);
  }
```

```
606
607   if (fighter_directory_exists(FIGHTER_DEFAULT_RYU_ROOT) &&
608       fighter_directory_exists(FIGHTER_DEFAULT_KEN_ROOT) &&
609       fighter_animation_system_init_with_roots(system,
610                                                FIGHTER_DEFAULT_RYU_ROOT,
611                                                FIGHTER_DEFAULT_KEN_ROOT) == 0) {
612     return 0;
613   }
614
615   return fighter_animation_system_init_with_roots(system,
616                                                   FIGHTER_REPO_RYU_ROOT,
617                                                   FIGHTER_REPO_KEN_ROOT);
618 }
619
620
621 int fighter_animation_system_init_with_roots(fighter_animation_system_t *system,
622                                              const char *ryu_root,
623                                              const char *ken_root) {
624   int i;
625
626   if (!system || !ryu_root || !ken_root) {
627     return -1;
628   }
629
630   memset(system, 0, sizeof(*system));
631
632   if (fighter_load_character_animation_set(ryu_root, &system->ryu) != 0) {
633     fighter_animation_system_close(system);
634     return -1;
635   }
636
637   if (fighter_load_character_animation_set(ken_root, &system->ken) != 0) {
638     fighter_animation_system_close(system);
639     return -1;
640   }
641
642   for (i = 0; i < FIGHTER_ANIMATION_MAX_PLAYERS; ++i) {
643     fighter_animation_state_reset(&system->players[i], NULL);
644   }
645
646   return 0;
647 }
648
649
650 void fighter_animation_system_close(fighter_animation_system_t *system) {
651   int i;
652
653   if (!system) {
654     return;
655   }
656
657   fighter_free_animation_set(&system->ryu);
658   fighter_free_animation_set(&system->ken);
659
660   for (i = 0; i < FIGHTER_ANIMATION_MAX_PLAYERS; ++i) {
661     fighter_animation_state_reset(&system->players[i], NULL);
662   }
663 }
664
665
666 void fighter_animation_system_update(fighter_animation_system_t *system,
667                                      const fighter_game_t *game) {
668   int i;
669
670   if (!system || !game) {
671     return;
672   }
673
674   for (i = 0; i < FIGHTER_ANIMATION_MAX_PLAYERS; ++i) {
675     const fighter_player_state_t *player = &game->players[i];
676     const fighter_character_animation_set_t *set =
677         fighter_select_character_set(system, player->character_id);
678     const fighter_animation_clip_t *next_clip =
```

```
        fighter_select_clip_for_player(player, set);
    fighter_player_animation_state_t *state = &system->players[i];

    if (state->current_clip != next_clip) {
      fighter_animation_state_reset(state, next_clip);
    } else {
      fighter_animation_state_advance(state);
    }
  }
}


const fighter_sprite_t *fighter_animation_current_sprite(
    const fighter_animation_system_t *system,
    int player_index) {
  const fighter_player_animation_state_t *state;
  const fighter_animation_clip_t *clip;

  if (!system || player_index < 0 ||
      player_index >= FIGHTER_ANIMATION_MAX_PLAYERS) {
    return NULL;
  }

  state = &system->players[player_index];
  clip = state->current_clip;
  if (!clip || clip->frame_count <= 0 || !clip->frames) {
    return NULL;
  }

  if (state->frame_index < 0 || state->frame_index >= clip->frame_count) {
    return NULL;
  }

  return &clip->frames[state->frame_index];
}


const fighter_animation_clip_t *fighter_animation_current_clip(
    const fighter_animation_system_t *system,
    int player_index) {
  if (!system || player_index < 0 ||
      player_index >= FIGHTER_ANIMATION_MAX_PLAYERS) {
    return NULL;
  }

  return system->players[player_index].current_clip;
}


int fighter_animation_current_frame_index(
    const fighter_animation_system_t *system,
    int player_index) {
  if (!system || player_index < 0 ||
      player_index >= FIGHTER_ANIMATION_MAX_PLAYERS) {
    return 0;
  }

  return system->players[player_index].frame_index;
}
```

A.1.11 sw/fighter_ui.c

```
#include "fighter_ui.h"

#include <stdio.h>

void fighter_ui_init(fighter_ui_context_t *ui) {
  if (ui == NULL) {
    return;
  }
```

```
9
10    ui->bg_frame = 0;
11    ui->frame_counter = 0;
12    ui->blink_on = 1;
13    ui->blink_counter = 0;
14  }
15
16  void fighter_ui_update(fighter_ui_context_t *ui) {
17    if (ui == NULL) {
18      return;
19    }
20
21
22    ui->frame_counter++;
23    if (ui->frame_counter >= 30) {
24      ui->bg_frame = !ui->bg_frame;
25      ui->frame_counter = 0;
26    }
27
28
29    ui->blink_counter++;
30    if (ui->blink_counter >= 20) {
31      ui->blink_on = !ui->blink_on;
32      ui->blink_counter = 0;
33    }
34  }
35
36  void fighter_ui_render_menu(const fighter_ui_context_t *ui,
37                              const fighter_menu_result_t *menu_result) {
38    if (ui == NULL || menu_result == NULL) {
39      return;
40    }
41
42
43    printf("[UI MENU] bg=%d blink=%d selected=%s action=%d\n",
44           ui->bg_frame,
45           ui->blink_on,
46           fighter_menu_item_name(menu_result->selected_item),
47           (int)menu_result->action);
48  }
49
50  void fighter_ui_render_battle(const fighter_ui_context_t *ui) {
51    if (ui == NULL) {
52      return;
53    }
54
55    printf("[UI BATTLE] bg=%d\n", ui->bg_frame);
56  }
```

A.1.12 sw/main_audio_demo.c

```
1   #include "fighter_audio.h"
2
3   #include <signal.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <string.h>
7   #include <time.h>
8
9   static volatile sig_atomic_t g_running = 1;
10
11  static void fighter_audio_demo_on_signal(int signal_number) {
12    (void)signal_number;
13    g_running = 0;
14  }
15
16  static void fighter_audio_demo_sleep_ms(long milliseconds) {
17    struct timespec delay;
18
19    if (milliseconds <= 0) {
```

```
    return;
  }

  delay.tv_sec = (time_t)(milliseconds / 1000L);
  delay.tv_nsec = (long)(milliseconds % 1000L) * 1000000L;
  nanosleep(&delay, NULL);
}

static void fighter_audio_demo_print_usage(const char *argv0) {
  printf("usage: %s [--track name] [--loop] [--seconds N | --forever] "
         "[--list]\n",
         argv0);
}

static void fighter_audio_demo_print_tracks(void) {
  puts("available tracks:");
  puts("  menu_bgm");
  puts("  menu_confirm");
  puts("  game_over");
}

static int fighter_audio_demo_parse_track(const char *name,
                                          fighter_audio_track_t *track_out) {
  if (!name || !track_out) {
    return -1;
  }

  if (strcmp(name, "menu_bgm") == 0) {
    *track_out = FIGHTER_AUDIO_TRACK_MENU_BGM;
    return 0;
  }
  if (strcmp(name, "menu_confirm") == 0) {
    *track_out = FIGHTER_AUDIO_TRACK_MENU_CONFIRM;
    return 0;
  }
  if (strcmp(name, "game_over") == 0) {
    *track_out = FIGHTER_AUDIO_TRACK_GAME_OVER;
    return 0;
  }

  return -1;
}

static int fighter_audio_demo_parse_seconds(const char *text, double *seconds_out) {
  char *end;
  double value;

  if (!text || !seconds_out) {
    return -1;
  }

  value = strtod(text, &end);
  if (end == text || !end || *end != '\0' || value <= 0.0) {
    return -1;
  }

  *seconds_out = value;
  return 0;
}

static double fighter_audio_demo_default_seconds(fighter_audio_track_t track,
                                                 int loop_enabled) {
  if (loop_enabled || track == FIGHTER_AUDIO_TRACK_MENU_BGM) {
    return 5.0;
  }

  return 3.0;
}

int main(int argc, char **argv) {
  fighter_audio_context_t context;
  fighter_audio_options_t options;
  fighter_audio_command_list_t commands;
```

```
  fighter_audio_track_t track = FIGHTER_AUDIO_TRACK_MENU_CONFIRM;
  double hold_seconds = 0.0;
  int loop_enabled = 0;
  int hold_forever = 0;
  int i;

  for (i = 1; i < argc; ++i) {
    if (strcmp(argv[i], "--track") == 0 && i + 1 < argc) {
      ++i;
      if (fighter_audio_demo_parse_track(argv[i], &track) != 0) {
        fprintf(stderr, "unknown track: %s\n", argv[i]);
        fighter_audio_demo_print_tracks();
        return 1;
      }
    } else if (strcmp(argv[i], "--loop") == 0) {
      loop_enabled = 1;
    } else if (strcmp(argv[i], "--seconds") == 0 && i + 1 < argc) {
      ++i;
      if (fighter_audio_demo_parse_seconds(argv[i], &hold_seconds) != 0) {
        fprintf(stderr, "invalid seconds: %s\n", argv[i]);
        return 1;
      }
    } else if (strcmp(argv[i], "--forever") == 0) {
      hold_forever = 1;
    } else if (strcmp(argv[i], "--list") == 0) {
      fighter_audio_demo_print_tracks();
      return 0;
    } else if (strcmp(argv[i], "--help") == 0) {
      fighter_audio_demo_print_usage(argv[0]);
      fighter_audio_demo_print_tracks();
      return 0;
    } else {
      fprintf(stderr, "unknown argument: %s\n", argv[i]);
      fighter_audio_demo_print_usage(argv[0]);
      return 1;
    }
  }

  if (!hold_forever && hold_seconds <= 0.0) {
    hold_seconds = fighter_audio_demo_default_seconds(track, loop_enabled);
  }

  signal(SIGINT, fighter_audio_demo_on_signal);
  signal(SIGTERM, fighter_audio_demo_on_signal);

  fighter_audio_options_init(&options);
  options.enable_command_audio = 1;

  if (fighter_audio_init(&context, &options) != 0) {
    fprintf(stderr, "audio init failed\n");
    return 1;
  }

  printf("audio backend: %s\n", fighter_audio_backend_name(&context));
  printf("track        : %s\n", fighter_audio_track_name(track));
  printf("mode         : %s\n", loop_enabled ? "loop" : "once");
  if (hold_forever) {
    printf("hold         : until Ctrl-C\n");
  } else {
    printf("hold         : %.2f s\n", hold_seconds);
  }

  if (context.backend == FIGHTER_AUDIO_BACKEND_DISABLED) {
    fighter_audio_close(&context);
    return 1;
  }

  fighter_audio_command_list_clear(&commands);
  if (fighter_audio_command_list_push(
          &commands,
          loop_enabled ? FIGHTER_AUDIO_COMMAND_START_LOOP
                       : FIGHTER_AUDIO_COMMAND_PLAY_ONCE,
          track) != 0) {
```

```
    fprintf(stderr, "failed to queue audio command\n");
    fighter_audio_close(&context);
    return 1;
  }
  fighter_audio_process_commands(&context, &commands);

  if (hold_forever) {
    while (g_running) {
      fighter_audio_demo_sleep_ms(50L);
    }
  } else {
    const long sleep_quantum_ms = 50L;
    long remaining_ms = (long)(hold_seconds * 1000.0);

    while (g_running && remaining_ms > 0) {
      long step_ms = remaining_ms < sleep_quantum_ms ? remaining_ms
                                                     : sleep_quantum_ms;
      fighter_audio_demo_sleep_ms(step_ms);
      remaining_ms -= step_ms;
    }
  }

  if (loop_enabled) {
    fighter_audio_command_list_clear(&commands);
    (void)fighter_audio_command_list_push(&commands,
                                          FIGHTER_AUDIO_COMMAND_STOP_LOOP,
                                          FIGHTER_AUDIO_TRACK_NONE);
    fighter_audio_process_commands(&context, &commands);
  }

  fighter_audio_close(&context);
  return 0;
}
```

A.1.13 sw/main_audio_probe.c

```
#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

enum {
  FIGHTER_AUDIO_MMIO_REG_COUNT = 4
};

static const off_t k_default_bridge_reset_addr = (off_t)0xFFD0501C;
static const off_t k_default_mmio_addr = (off_t)0xFF200000;

static int fighter_audio_probe_parse_address(const char *text,
                                             off_t default_value,
                                             off_t *value_out) {
  char *end;
  unsigned long long value;

  if (!value_out) {
    return -1;
  }

  if (!text || text[0] == '\0') {
    *value_out = default_value;
    return 0;
  }

  errno = 0;
  value = strtoull(text, &end, 0);
```

```
  if (errno != 0 || end == text || !end || *end != '\0') {
    return -1;
  }

  *value_out = (off_t)value;
  return 0;
}

static int fighter_audio_probe_parse_env_or_default(const char *env_name,
                                                    off_t default_value,
                                                    off_t *value_out) {
  const char *text;

  text = getenv(env_name);
  return fighter_audio_probe_parse_address(text, default_value, value_out);
}

static int fighter_audio_probe_map_region(int mem_fd,
                                          off_t physical_addr,
                                          size_t span,
                                          void **map_base,
                                          size_t *map_length,
                                          volatile uint32_t **register_base) {
  long page_size;
  off_t page_base;
  off_t page_offset;
  size_t length;
  void *mapped;

  if (mem_fd < 0 || !map_base || !map_length || !register_base || span == 0) {
    return -1;
  }

  page_size = sysconf(_SC_PAGESIZE);
  if (page_size <= 0) {
    return -1;
  }

  page_base = physical_addr & ~((off_t)page_size - 1);
  page_offset = physical_addr - page_base;
  length = (size_t)page_offset + span;
  length = (length + (size_t)page_size - 1U) & ~((size_t)page_size - 1U);

  mapped = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd,
                page_base);
  if (mapped == MAP_FAILED) {
    return -1;
  }

  *map_base = mapped;
  *map_length = length;
  *register_base =
      (volatile uint32_t *)((unsigned char *)mapped + (size_t)page_offset);
  return 0;
}

static void fighter_audio_probe_unmap_region(void **map_base,
                                             size_t *map_length) {
  if (!map_base || !map_length || !*map_base || *map_length == 0) {
    return;
  }

  munmap(*map_base, *map_length);
  *map_base = NULL;
  *map_length = 0;
}

static void fighter_audio_probe_print_usage(const char *argv0) {
  printf("usage: %s [--mmio-addr HEX] [--bridge-addr HEX] [--no-enable-bridge]\n",
         argv0);
}

int main(int argc, char **argv) {
```

```
  off_t mmio_addr;
  off_t bridge_addr;
  int enable_bridge = 1;
  int mem_fd = -1;
  void *bridge_map = NULL;
  size_t bridge_map_length = 0;
  volatile uint32_t *bridge_reg = NULL;
  void *audio_map = NULL;
  size_t audio_map_length = 0;
  volatile uint32_t *audio_regs = NULL;
  uint32_t bridge_before = 0;
  uint32_t bridge_after = 0;
  uint32_t fifospace;
  int i;

  if (fighter_audio_probe_parse_env_or_default("FIGHTER_AUDIO_MMIO_ADDR",
                                               k_default_mmio_addr,
                                               &mmio_addr) != 0) {
    fprintf(stderr, "invalid FIGHTER_AUDIO_MMIO_ADDR\n");
    return 1;
  }
  if (fighter_audio_probe_parse_env_or_default("FIGHTER_AUDIO_BRIDGE_RESET_ADDR",
                                               k_default_bridge_reset_addr,
                                               &bridge_addr) != 0) {
    fprintf(stderr, "invalid FIGHTER_AUDIO_BRIDGE_RESET_ADDR\n");
    return 1;
  }

  for (i = 1; i < argc; ++i) {
    if (strcmp(argv[i], "--mmio-addr") == 0 && i + 1 < argc) {
      ++i;
      if (fighter_audio_probe_parse_address(argv[i], mmio_addr, &mmio_addr) != 0) {
        fprintf(stderr, "invalid --mmio-addr: %s\n", argv[i]);
        return 1;
      }
    } else if (strcmp(argv[i], "--bridge-addr") == 0 && i + 1 < argc) {
      ++i;
      if (fighter_audio_probe_parse_address(argv[i], bridge_addr, &bridge_addr) !=
          0) {
        fprintf(stderr, "invalid --bridge-addr: %s\n", argv[i]);
        return 1;
      }
    } else if (strcmp(argv[i], "--no-enable-bridge") == 0) {
      enable_bridge = 0;
    } else if (strcmp(argv[i], "--help") == 0) {
      fighter_audio_probe_print_usage(argv[0]);
      return 0;
    } else {
      fprintf(stderr, "unknown argument: %s\n", argv[i]);
      fighter_audio_probe_print_usage(argv[0]);
      return 1;
    }
  }

  mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (mem_fd < 0) {
    fprintf(stderr, "open /dev/mem failed: %s\n", strerror(errno));
    return 1;
  }

  if (fighter_audio_probe_map_region(mem_fd, bridge_addr, sizeof(uint32_t),
                                     &bridge_map, &bridge_map_length,
                                     &bridge_reg) != 0) {
    fprintf(stderr, "map bridge reset 0x%08lX failed: %s\n",
            (unsigned long)bridge_addr, strerror(errno));
    close(mem_fd);
    return 1;
  }

  if (fighter_audio_probe_map_region(mem_fd, mmio_addr,
                                     FIGHTER_AUDIO_MMIO_REG_COUNT *
                                         sizeof(uint32_t),
                                     &audio_map, &audio_map_length,
```

```
                                    &audio_regs) != 0) {
    fprintf(stderr, "map audio mmio 0x%08lX failed: %s\n",
            (unsigned long)mmio_addr, strerror(errno));
    fighter_audio_probe_unmap_region(&bridge_map, &bridge_map_length);
    close(mem_fd);
    return 1;
  }

  bridge_before = *bridge_reg;
  bridge_after = bridge_before;
  if (enable_bridge) {
    bridge_after &= ~0x3U;
    *bridge_reg = bridge_after;
    bridge_after = *bridge_reg;
  }

  printf("bridge_reset_addr : 0x%08lX\n", (unsigned long)bridge_addr);
  printf("audio_mmio_addr   : 0x%08lX\n", (unsigned long)mmio_addr);
  printf("bridge_before     : 0x%08" PRIX32 "\n", bridge_before);
  printf("bridge_after      : 0x%08" PRIX32 "\n", bridge_after);

  for (i = 0; i < FIGHTER_AUDIO_MMIO_REG_COUNT; ++i) {
    printf("reg[%d]            : 0x%08" PRIX32 "\n", i, audio_regs[i]);
  }

  fifospace = audio_regs[1];
  printf("left_write_space  : %" PRIu32 "\n", (fifospace >> 24) & 0xFFU);
  printf("right_write_space : %" PRIu32 "\n", (fifospace >> 16) & 0xFFU);

  fighter_audio_probe_unmap_region(&audio_map, &audio_map_length);
  fighter_audio_probe_unmap_region(&bridge_map, &bridge_map_length);
  close(mem_fd);
  return 0;
}
```

A.1.14 sw/main_gamepad_probe.c

```
#include <errno.h>
#include <fcntl.h>
#include <linux/input.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>

static const char *event_type_name(unsigned short type) {
  switch (type) {
    case EV_SYN:
      return "EV_SYN";
    case EV_KEY:
      return "EV_KEY";
    case EV_REL:
      return "EV_REL";
    case EV_ABS:
      return "EV_ABS";
    default:
      return "EV_OTHER";
  }
}

static const char *key_code_name(unsigned short code) {
  switch (code) {
    case BTN_A:
      return "BTN_A";
    case BTN_B:
      return "BTN_B";
    case BTN_X:
      return "BTN_X";
    case BTN_Y:
```

```
      return "BTN_Y";
    case BTN_TL:
      return "BTN_TL";
    case BTN_TR:
      return "BTN_TR";
    case BTN_TL2:
      return "BTN_TL2";
    case BTN_TR2:
      return "BTN_TR2";
    case BTN_SELECT:
      return "BTN_SELECT";
    case BTN_START:
      return "BTN_START";
    case BTN_MODE:
      return "BTN_MODE";
    case BTN_THUMBL:
      return "BTN_THUMBL";
    case BTN_THUMBR:
      return "BTN_THUMBR";
    case BTN_TRIGGER:
      return "BTN_TRIGGER";
    case BTN_THUMB:
      return "BTN_THUMB";
    case BTN_THUMB2:
      return "BTN_THUMB2";
    case BTN_TOP:
      return "BTN_TOP";
    case BTN_TOP2:
      return "BTN_TOP2";
    case BTN_PINKIE:
      return "BTN_PINKIE";
    case BTN_BASE:
      return "BTN_BASE";
    case BTN_BASE2:
      return "BTN_BASE2";
    case BTN_BASE3:
      return "BTN_BASE3";
    case BTN_BASE4:
      return "BTN_BASE4";
    case BTN_BASE5:
      return "BTN_BASE5";
    case BTN_BASE6:
      return "BTN_BASE6";
    case BTN_DPAD_UP:
      return "BTN_DPAD_UP";
    case BTN_DPAD_DOWN:
      return "BTN_DPAD_DOWN";
    case BTN_DPAD_LEFT:
      return "BTN_DPAD_LEFT";
    case BTN_DPAD_RIGHT:
      return "BTN_DPAD_RIGHT";
    default:
      return "KEY_OTHER";
  }
}

static const char *abs_code_name(unsigned short code) {
  switch (code) {
    case ABS_X:
      return "ABS_X";
    case ABS_Y:
      return "ABS_Y";
    case ABS_Z:
      return "ABS_Z";
    case ABS_RX:
      return "ABS_RX";
    case ABS_RY:
      return "ABS_RY";
    case ABS_RZ:
      return "ABS_RZ";
    case ABS_HAT0X:
      return "ABS_HAT0X";
    case ABS_HAT0Y:
```

```
      return "ABS_HAT0Y";
    case ABS_HAT1X:
      return "ABS_HAT1X";
    case ABS_HAT1Y:
      return "ABS_HAT1Y";
    default:
      return "ABS_OTHER";
  }
}

static const char *event_code_name(unsigned short type, unsigned short code) {
  if (type == EV_KEY) {
    return key_code_name(code);
  }
  if (type == EV_ABS) {
    return abs_code_name(code);
  }
  return "-";
}

static void print_usage(const char *argv0) {
  printf("usage: %s [event-device]\n", argv0);
  printf("default: /dev/input/by-id/usb-081f_USB_gamepad-event-joystick\n");
}

int main(int argc, char **argv) {
  const char *device_path = "/dev/input/by-id/usb-081f_USB_gamepad-event-joystick";
  char device_name[256];
  int fd;

  if (argc > 2) {
    print_usage(argv[0]);
    return 1;
  }
  if (argc == 2) {
    if (strcmp(argv[1], "--help") == 0) {
      print_usage(argv[0]);
      return 0;
    }
    device_path = argv[1];
  }

  fd = open(device_path, O_RDONLY);
  if (fd < 0) {
    fprintf(stderr, "open %s failed: %s\n", device_path, strerror(errno));
    return 1;
  }

  memset(device_name, 0, sizeof(device_name));
  if (ioctl(fd, EVIOCGNAME(sizeof(device_name) - 1), device_name) < 0) {
    snprintf(device_name, sizeof(device_name), "unknown");
  }

  setvbuf(stdout, NULL, _IOLBF, 0);
  printf("device: %s\n", device_path);
  printf("name  : %s\n", device_name);
  printf("press buttons / d-pad / sticks; Ctrl-C to stop\n");
  printf("type,code,value,name\n");

  while (1) {
    struct input_event event;
    ssize_t bytes_read = read(fd, &event, sizeof(event));

    if (bytes_read < 0) {
      if (errno == EINTR) {
        continue;
      }
      fprintf(stderr, "read failed: %s\n", strerror(errno));
      close(fd);
      return 1;
    }
    if (bytes_read != (ssize_t)sizeof(event)) {
      fprintf(stderr, "short read: %zd bytes\n", bytes_read);
```

```
      close(fd);
      return 1;
    }

    if (event.type == EV_SYN) {
      continue;
    }

    printf("%s,%u,%d,%s\n",
           event_type_name(event.type),
           event.code,
           event.value,
           event_code_name(event.type, event.code));
  }
}
```

A.1.15 sw/main_input_demo.c

```
#include "fighter_input.h"
#include "usb_hid_keyboard.h"
#include "fighter_ui.h"

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

static volatile sig_atomic_t g_running = 1;

static void on_signal(int signal_number) {
  (void)signal_number;
  g_running = 0;
}

static void sleep_for_poll_interval(void) {
  struct timespec delay;

  delay.tv_sec = 0;
  delay.tv_nsec = 1000000L;
  nanosleep(&delay, NULL);
}

static int player_result_changed(const fighter_player_result_t *lhs,
                                 const fighter_player_result_t *rhs) {
  return memcmp(lhs, rhs, sizeof(*lhs)) != 0;
}

static void print_player_result(int player_index,
                                const fighter_player_result_t *result) {
  printf("P%d: left=%d right=%d jump=%d crouch=%d guard=%d attack=%s exit=%d\n",
         player_index + 1,
         result->move_left,
         result->move_right,
         result->jump_held,
         result->crouch_held,
         result->guard_held,
         fighter_attack_command_name(result->attack_command),
         result->exit_requested);
}

int main(void) {
  usb_hid_keyboard_manager_t keyboard_manager;
  usb_hid_keyboard_report_t reports[USB_HID_KEYBOARD_MAX_DEVICES];
  fighter_menu_parser_t menu_parser;
  fighter_player_parser_t player_parsers[USB_HID_KEYBOARD_MAX_DEVICES];
  fighter_player_result_t previous_results[USB_HID_KEYBOARD_MAX_DEVICES];
  fighter_menu_result_t menu_result;
  fighter_ui_context_t ui;

```

```
53  int in_menu = 1;
54  int rc;
55  size_t i;
56
57  signal(SIGINT, on_signal);
58  signal(SIGTERM, on_signal);
59
60  memset(previous_results, 0, sizeof(previous_results));
61  memset(&menu_result, 0, sizeof(menu_result));
62
63  fighter_menu_parser_init(&menu_parser);
64  fighter_ui_init(&ui);
65
66  for (i = 0; i < USB_HID_KEYBOARD_MAX_DEVICES; ++i) {
67    fighter_player_parser_init(&player_parsers[i]);
68  }
69
70  rc = usb_hid_keyboard_manager_init(&keyboard_manager, USB_HID_KEYBOARD_MAX_DEVICES);
71  if (rc != 0) {
72    fprintf(stderr, "failed to open USB keyboard(s): %d\n", rc);
73    return 1;
74  }
75
76  printf("opened %zu keyboard(s)\n", keyboard_manager.device_count);
77  for (i = 0; i < keyboard_manager.device_count; ++i) {
78    const usb_hid_keyboard_device_t *device = &keyboard_manager.devices[i];
79    printf("  keyboard %zu -> %s (bus=%d addr=%d)\n",
80           i + 1,
81           device->product_name,
82           device->bus_number,
83           device->device_address);
84  }
85
86  printf("\n");
87  printf("menu controls on keyboard 1:\n");
88  printf("  A/D switch, J confirm\n");
89  printf("battle controls:\n");
90  printf("  each keyboard uses W/A/S/D/J/K/L\n");
91  printf("  P1 = keyboard 1, P2 = keyboard 2\n");
92  printf("  L returns to menu in this demo\n");
93  printf("\n");
94
95  while (g_running) {
96    memset(reports, 0, sizeof(reports));
97    rc = usb_hid_keyboard_manager_poll(&keyboard_manager,
98                                       reports,
99                                       USB_HID_KEYBOARD_MAX_DEVICES,
100                                      8);
101   if (rc < 0) {
102     fprintf(stderr, "poll failed: %d\n", rc);
103     break;
104   }
105
106
107   fighter_ui_update(&ui);
108
109   if (in_menu) {
110     fighter_menu_parser_update(&menu_parser, &reports[0], &menu_result);
111
112     if (menu_result.action == FIGHTER_MENU_ACTION_MOVE_LEFT ||
113         menu_result.action == FIGHTER_MENU_ACTION_MOVE_RIGHT) {
114       printf("menu selection -> %s\n",
115              fighter_menu_item_name(menu_result.selected_item));
116
117
118     } else if (menu_result.action == FIGHTER_MENU_ACTION_CONFIRM) {
119       printf("menu confirm -> %s\n",
120              fighter_menu_item_name(menu_result.selected_item));
121
122
123       if (menu_result.selected_item == FIGHTER_MENU_ITEM_START) {
124         in_menu = 0;
125         memset(previous_results, 0, sizeof(previous_results));
```

```
126           for (i = 0; i < USB_HID_KEYBOARD_MAX_DEVICES; ++i) {
127             fighter_player_parser_init(&player_parsers[i]);
128           }
129           printf("enter battle mode\n");
130         } else {
131           printf("exit selected\n");
132           break;
133         }
134       }
135
136       fighter_ui_render_menu(&ui, &menu_result);
137     } else {
138       fighter_ui_render_battle(&ui);
139
140       for (i = 0; i < keyboard_manager.device_count; ++i) {
141         fighter_player_result_t result;
142
143         fighter_player_parser_update(&player_parsers[i], &reports[i], &result);
144
145         if (result.exit_requested) {
146           printf("P%zu requested exit, return to menu\n", i + 1);
147           fighter_menu_parser_init(&menu_parser);
148           in_menu = 1;
149           break;
150         }
151
152         if (player_result_changed(&result, &previous_results[i]) ||
153             result.attack_command != FIGHTER_ATTACK_NONE) {
154           print_player_result((int)i, &result);
155           previous_results[i] = result;
156         }
157       }
158     }
159     sleep_for_poll_interval();
160   }
161
162   usb_hid_keyboard_manager_close(&keyboard_manager);
163   return 0;
164 }
```

A.1.16 sw/main_phase1_demo.c

```
1  #include "fighter_audio.h"
2  #include "fighter_game.h"
3  #include "fighter_gamepad.h"
4  #include "fighter_input.h"
5  #include "fighter_renderer.h"
6  #include "fighter_animation.h"
7
8  #include <errno.h>
9  #include <signal.h>
10 #include <stdint.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <time.h>
15
16 #if FIGHTER_ENABLE_LIBUSB
17 #  include "usb_hid_keyboard.h"
18 #endif
19
20 static volatile sig_atomic_t g_running = 1;
21
22 typedef enum {
23   FIGHTER_INPUT_MODE_SCRIPT = 0,
24   FIGHTER_INPUT_MODE_USB = 1,
25   FIGHTER_INPUT_MODE_GAMEPAD = 2
26 } fighter_input_mode_t;
27
28 typedef enum {
```

```
  FIGHTER_SCRIPT_SMOKE = 0,
  FIGHTER_SCRIPT_KO = 1
} fighter_script_kind_t;

static void fighter_on_signal(int signal_number) {
  (void)signal_number;
  g_running = 0;
}

static int64_t fighter_now_ns(void) {
  struct timespec now;

  if (clock_gettime(CLOCK_MONOTONIC, &now) != 0) {
    return 0;
  }

  return (int64_t)now.tv_sec * 1000000000LL + (int64_t)now.tv_nsec;
}

static void fighter_sleep_ns(int64_t duration_ns) {
  struct timespec delay;

  if (duration_ns <= 0) {
    return;
  }

  delay.tv_sec = (time_t)(duration_ns / 1000000000LL);
  delay.tv_nsec = (long)(duration_ns % 1000000000LL);
  nanosleep(&delay, NULL);
}

static void fighter_build_script_reports(fighter_script_kind_t kind,
                                         int frame_index,
                                         usb_hid_keyboard_report_t reports[2]) {
  usb_hid_keyboard_report_clear(&reports[0]);
  usb_hid_keyboard_report_clear(&reports[1]);

  if (kind == FIGHTER_SCRIPT_SMOKE) {
    if (frame_index == 10) {
      (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_J);
    } else if (frame_index >= 25 && frame_index <= 60) {
      (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_D);
    } else if (frame_index == 70) {
      (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_J);
    } else if (frame_index >= 85 && frame_index <= 120) {
      (void)usb_hid_keyboard_report_add_key(&reports[1], FIGHTER_HID_KEY_A);
    } else if (frame_index == 130) {
      (void)usb_hid_keyboard_report_add_key(&reports[1], FIGHTER_HID_KEY_J);
    } else if (frame_index == 170) {
      (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_L);
    }
    return;
  }

  if (frame_index == 8) {
    (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_J);
    return;
  }
  if (frame_index >= 20 && frame_index <= 50) {
    (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_D);
    (void)usb_hid_keyboard_report_add_key(&reports[1], FIGHTER_HID_KEY_A);
    return;
  }
  if (frame_index == 65 || frame_index == 85 || frame_index == 105 ||
      frame_index == 125 || frame_index == 145 || frame_index == 165) {
    (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_D);
    (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_J);
    return;
  }
  if (frame_index == 310) {
    (void)usb_hid_keyboard_report_add_key(&reports[0], FIGHTER_HID_KEY_J);
  }
}
```

```
102
103 static const char *fighter_script_name(fighter_script_kind_t kind) {
104   switch (kind) {
105     case FIGHTER_SCRIPT_KO:
106       return "ko";
107     case FIGHTER_SCRIPT_SMOKE:
108     default:
109       return "smoke";
110   }
111 }
112
113 static const char *fighter_input_mode_name(fighter_input_mode_t mode,
114                                            fighter_script_kind_t script_kind) {
115   switch (mode) {
116     case FIGHTER_INPUT_MODE_USB:
117       return "usb_keyboard";
118     case FIGHTER_INPUT_MODE_GAMEPAD:
119       return "gamepad";
120     case FIGHTER_INPUT_MODE_SCRIPT:
121     default:
122       return fighter_script_name(script_kind);
123   }
124 }
125
126 static void fighter_print_usage(const char *argv0) {
127   printf("usage: %s [--usb] [--gamepad] [--gamepad0 PATH] [--gamepad1 PATH] "
128          "[--script smoke|ko] [--console] [--fb PATH] [--audio] [--frames N]\n",
129          argv0);
130 }
131
132 int main(int argc, char **argv) {
133   const int64_t k_logic_tick_ns = 16666667LL;
134   const int64_t k_max_accumulator_ns = 250000000LL;
135   fighter_game_t game;
136   fighter_renderer_t renderer;
137   fighter_renderer_options_t renderer_options;
138   fighter_audio_context_t audio_context;
139   fighter_audio_options_t audio_options;
140   fighter_animation_system_t anim_system;
141   fighter_player_parser_t parsers[FIGHTER_PLAYER_COUNT];
142   fighter_gamepad_t gamepads[FIGHTER_PLAYER_COUNT];
143   fighter_audio_command_list_t audio_commands;
144   fighter_input_mode_t input_mode;
145   fighter_script_kind_t script_kind;
146   const char *gamepad_paths[FIGHTER_PLAYER_COUNT];
147   int max_frames;
148   int frame_index;
149   int realtime;
150   int use_fixed_timestep;
151   int i;
152   int64_t previous_tick_ns;
153   int64_t accumulator_ns;
154 #if FIGHTER_ENABLE_LIBUSB
155   usb_hid_keyboard_manager_t keyboard_manager;
156   int keyboard_manager_ready;
157 #endif
158
159   input_mode = FIGHTER_INPUT_MODE_SCRIPT;
160   script_kind = FIGHTER_SCRIPT_KO;
161   gamepad_paths[0] = "/dev/input/event0";
162   gamepad_paths[1] = "/dev/input/event1";
163   max_frames = 420;
164   realtime = 0;
165
166   fighter_renderer_options_init(&renderer_options);
167   fighter_audio_options_init(&audio_options);
168
169   for (i = 1; i < argc; ++i) {
170     if (strcmp(argv[i], "--usb") == 0) {
171       input_mode = FIGHTER_INPUT_MODE_USB;
172       max_frames = -1;
173       realtime = 1;
174     } else if (strcmp(argv[i], "--gamepad") == 0) {
```

```
      input_mode = FIGHTER_INPUT_MODE_GAMEPAD;
      max_frames = -1;
      realtime = 1;
    } else if (strcmp(argv[i], "--gamepad0") == 0 && i + 1 < argc) {
      gamepad_paths[0] = argv[++i];
      input_mode = FIGHTER_INPUT_MODE_GAMEPAD;
      max_frames = -1;
      realtime = 1;
    } else if (strcmp(argv[i], "--gamepad1") == 0 && i + 1 < argc) {
      gamepad_paths[1] = argv[++i];
      input_mode = FIGHTER_INPUT_MODE_GAMEPAD;
      max_frames = -1;
      realtime = 1;
    } else if (strcmp(argv[i], "--script") == 0 && i + 1 < argc) {
      ++i;
      if (strcmp(argv[i], "smoke") == 0) {
        script_kind = FIGHTER_SCRIPT_SMOKE;
        max_frames = 220;
      } else if (strcmp(argv[i], "ko") == 0) {
        script_kind = FIGHTER_SCRIPT_KO;
        max_frames = 420;
      } else {
        fprintf(stderr, "unknown script: %s\n", argv[i]);
        fighter_print_usage(argv[0]);
        return 1;
      }
    } else if (strcmp(argv[i], "--console") == 0) {
      renderer_options.prefer_framebuffer = 0;
    } else if (strcmp(argv[i], "--fb") == 0 && i + 1 < argc) {
      renderer_options.framebuffer_path = argv[++i];
      renderer_options.prefer_framebuffer = 1;
    } else if (strcmp(argv[i], "--audio") == 0) {
      audio_options.enable_command_audio = 1;
    } else if (strcmp(argv[i], "--frames") == 0 && i + 1 < argc) {
      ++i;
      max_frames = atoi(argv[i]);
    } else if (strcmp(argv[i], "--realtime") == 0) {
      realtime = 1;
    } else if (strcmp(argv[i], "--help") == 0) {
      fighter_print_usage(argv[0]);
      return 0;
    } else {
      fprintf(stderr, "unknown argument: %s\n", argv[i]);
      fighter_print_usage(argv[0]);
      return 1;
    }
  }

  signal(SIGINT, fighter_on_signal);
  signal(SIGTERM, fighter_on_signal);

  fighter_game_init(&game, NULL);
  (void)fighter_renderer_init(&renderer, &renderer_options);
  (void)fighter_audio_init(&audio_context, &audio_options);

  if (fighter_animation_system_init(&anim_system) != 0) {
    fprintf(stderr, "failed to initialize animation system\n");
    fighter_renderer_close(&renderer);
    fighter_audio_close(&audio_context);
    return 1;
  }

  for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
    fighter_player_parser_init(&parsers[i]);
    fighter_gamepad_init(&gamepads[i]);
  }

  if (input_mode == FIGHTER_INPUT_MODE_GAMEPAD) {
    for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
      if (fighter_gamepad_open(&gamepads[i], gamepad_paths[i]) != 0) {
        fprintf(stderr, "failed to open gamepad %d at %s: %s\n", i + 1,
                gamepad_paths[i], strerror(errno));
        if (i == 0) {
```

```
          fighter_animation_system_close(&anim_system);
          fighter_renderer_close(&renderer);
          fighter_audio_close(&audio_context);
          return 1;
        }
      }
    }
  }

#if FIGHTER_ENABLE_LIBUSB
  keyboard_manager_ready = 0;
  memset(&keyboard_manager, 0, sizeof(keyboard_manager));
  if (input_mode == FIGHTER_INPUT_MODE_USB) {
    if (usb_hid_keyboard_manager_init(&keyboard_manager, FIGHTER_PLAYER_COUNT) != 0) {
      fprintf(stderr, "failed to open USB keyboard(s), falling back to script mode\n");
      input_mode = FIGHTER_INPUT_MODE_SCRIPT;
    } else {
      keyboard_manager_ready = 1;
    }
  }
#else
  if (input_mode == FIGHTER_INPUT_MODE_USB) {
    fprintf(stderr,
            "this build was compiled without libusb support; use --script or "
            "install libusb on the target board\n");
    fighter_animation_system_close(&anim_system);
    fighter_renderer_close(&renderer);
    fighter_audio_close(&audio_context);
    return 1;
  }
#endif

  printf("phase1 demo starting\n");
  printf("  input mode: %s\n",
         fighter_input_mode_name(input_mode, script_kind));
  if (input_mode == FIGHTER_INPUT_MODE_GAMEPAD) {
    for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
      printf("  gamepad %d : %s%s\n", i + 1, gamepad_paths[i],
             gamepads[i].connected ? "" : " (unavailable)");
    }
  }
  printf("  renderer  : %s\n", fighter_renderer_backend_name(&renderer));
  if (strcmp(fighter_renderer_backend_name(&renderer), "mmio") == 0) {
    printf("  detail    : %s\n", fighter_renderer_status_detail(&renderer));
  } else if (strcmp(fighter_renderer_backend_name(&renderer), "framebuffer") == 0) {
    printf("  framebuffer: %s\n",
           fighter_renderer_active_framebuffer_path(&renderer));
  } else if (renderer_options.prefer_framebuffer) {
    fprintf(stderr, "warning: VGA/framebuffer renderer unavailable, fallback to console\n");
    fprintf(stderr, "         detail: %s\n",
            fighter_renderer_status_detail(&renderer));
  }
  printf("  audio     : %s\n", fighter_audio_backend_name(&audio_context));

  use_fixed_timestep =
      realtime || strcmp(fighter_renderer_backend_name(&renderer), "mmio") == 0 ||
      strcmp(fighter_renderer_backend_name(&renderer), "framebuffer") == 0;
  frame_index = 0;
  previous_tick_ns = fighter_now_ns();
  accumulator_ns = 0;
  while (g_running && (max_frames < 0 || frame_index < max_frames)) {
    usb_hid_keyboard_report_t reports[FIGHTER_PLAYER_COUNT];
    fighter_player_result_t step_inputs[FIGHTER_PLAYER_COUNT];
    int logic_steps;

    logic_steps = 0;
    if (use_fixed_timestep) {
      int64_t now_ns = fighter_now_ns();
      int64_t elapsed_ns = now_ns - previous_tick_ns;

      if (elapsed_ns < 0) {
        elapsed_ns = 0;
      }
```

```
      if (elapsed_ns > k_max_accumulator_ns) {
        elapsed_ns = k_max_accumulator_ns;
      }

      accumulator_ns += elapsed_ns;
      if (accumulator_ns > k_max_accumulator_ns) {
        accumulator_ns = k_max_accumulator_ns;
      }
      previous_tick_ns = now_ns;

      if (accumulator_ns < k_logic_tick_ns) {
        fighter_sleep_ns(k_logic_tick_ns - accumulator_ns);
        continue;
      }
    }

    if (use_fixed_timestep && input_mode == FIGHTER_INPUT_MODE_GAMEPAD) {
      for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
        memset(&step_inputs[i], 0, sizeof(step_inputs[i]));
        if (gamepads[i].connected &&
            fighter_gamepad_update(&gamepads[i], &step_inputs[i]) != 0) {
          fprintf(stderr, "gamepad %d poll failed, stopping demo\n", i + 1);
          g_running = 0;
          break;
        }
      }
      if (!g_running) {
        break;
      }
    } else if (use_fixed_timestep && input_mode == FIGHTER_INPUT_MODE_USB) {
      for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
        usb_hid_keyboard_report_clear(&reports[i]);
      }

#if FIGHTER_ENABLE_LIBUSB
      if (usb_hid_keyboard_manager_poll(&keyboard_manager, reports,
                                        FIGHTER_PLAYER_COUNT, 8) < 0) {
        fprintf(stderr, "USB poll failed, stopping demo\n");
        break;
      }
#endif
      for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
        memset(&step_inputs[i], 0, sizeof(step_inputs[i]));
        fighter_player_parser_update(&parsers[i], &reports[i], &step_inputs[i]);
      }
    }

    while (g_running &&
           (max_frames < 0 || frame_index < max_frames) &&
           ((!use_fixed_timestep && logic_steps == 0) ||
            (use_fixed_timestep && accumulator_ns >= k_logic_tick_ns))) {
      if (!use_fixed_timestep || input_mode == FIGHTER_INPUT_MODE_SCRIPT) {
        for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
          usb_hid_keyboard_report_clear(&reports[i]);
          memset(&step_inputs[i], 0, sizeof(step_inputs[i]));
        }

        if (input_mode == FIGHTER_INPUT_MODE_SCRIPT) {
          fighter_build_script_reports(script_kind, frame_index, reports);
        } else {
#if FIGHTER_ENABLE_LIBUSB
          if (usb_hid_keyboard_manager_poll(&keyboard_manager, reports,
                                            FIGHTER_PLAYER_COUNT, 8) < 0) {
            fprintf(stderr, "USB poll failed, stopping demo\n");
            g_running = 0;
            break;
          }
#endif
        }

        for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
          fighter_player_parser_update(&parsers[i], &reports[i], &step_inputs[i]);
        }
```

```
394       }
395
396       if (!use_fixed_timestep && input_mode == FIGHTER_INPUT_MODE_GAMEPAD) {
397         for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
398           memset(&step_inputs[i], 0, sizeof(step_inputs[i]));
399           if (gamepads[i].connected &&
400               fighter_gamepad_update(&gamepads[i], &step_inputs[i]) != 0) {
401             fprintf(stderr, "gamepad %d poll failed, stopping demo\n", i + 1);
402             g_running = 0;
403             break;
404           }
405         }
406         if (!g_running) {
407           break;
408         }
409       }
410
411       fighter_game_tick(&game, step_inputs, &audio_commands);
412       fighter_animation_system_update(&anim_system, &game);
413       fighter_audio_process_commands(&audio_context, &audio_commands);
414
415       if (use_fixed_timestep) {
416         accumulator_ns -= k_logic_tick_ns;
417       }
418
419       ++frame_index;
420       logic_steps++;
421     }
422
423     if (logic_steps > 0) {
424       fighter_renderer_draw(&renderer, &game, &anim_system);
425     }
426   }
427
428 #if FIGHTER_ENABLE_LIBUSB
429   if (keyboard_manager_ready) {
430     usb_hid_keyboard_manager_close(&keyboard_manager);
431   }
432 #endif
433   for (i = 0; i < FIGHTER_PLAYER_COUNT; ++i) {
434     fighter_gamepad_close(&gamepads[i]);
435   }
436
437   fighter_animation_system_close(&anim_system);
438   fighter_audio_close(&audio_context);
439   fighter_renderer_close(&renderer);
440   return 0;
441 }
```

A.1.17 sw/main_vga_probe.c

```
1  #include <errno.h>
2  #include <fcntl.h>
3  #include <inttypes.h>
4  #include <stdint.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <sys/mman.h>
9  #include <unistd.h>
10
11 #include "fighter_vga_mmio.h"
12
13 static const off_t k_default_bridge_reset_addr = (off_t)0xFFD0501C;
14 static const off_t k_default_vga_mmio_addr = (off_t)0xFF240000;
15 static const uint32_t k_vga_ident = 0x56504741U;
16
17 static int parse_address(const char *text, off_t default_value, off_t *out) {
18   char *end;
19   unsigned long long value;
```

```
20
21   if (!out) {
22     return -1;
23   }
24   if (!text || text[0] == '\0') {
25     *out = default_value;
26     return 0;
27   }
28
29   errno = 0;
30   value = strtoull(text, &end, 0);
31   if (errno != 0 || end == text || !end || *end != '\0') {
32     return -1;
33   }
34
35   *out = (off_t)value;
36   return 0;
37 }
38
39 static int map_physical(int mem_fd,
40                         off_t physical_addr,
41                         size_t span,
42                         void **map_base,
43                         size_t *map_length,
44                         volatile uint32_t **register_base) {
45   long page_size;
46   off_t page_base;
47   off_t page_offset;
48   size_t length;
49   void *mapped;
50
51   page_size = sysconf(_SC_PAGESIZE);
52   if (mem_fd < 0 || page_size <= 0 || !map_base || !map_length ||
53       !register_base || span == 0) {
54     return -1;
55   }
56
57   page_base = physical_addr & ~((off_t)page_size - 1);
58   page_offset = physical_addr - page_base;
59   length = (size_t)page_offset + span;
60   length = (length + (size_t)page_size - 1U) & ~((size_t)page_size - 1U);
61
62   mapped = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd,
63                 page_base);
64   if (mapped == MAP_FAILED) {
65     return -1;
66   }
67
68   *map_base = mapped;
69   *map_length = length;
70   *register_base =
71       (volatile uint32_t *)((unsigned char *)mapped + (size_t)page_offset);
72   return 0;
73 }
74
75 static void unmap_region(void **map_base, size_t *map_length) {
76   if (!map_base || !map_length || !*map_base || *map_length == 0) {
77     return;
78   }
79
80   munmap(*map_base, *map_length);
81   *map_base = NULL;
82   *map_length = 0;
83 }
84
85 static void print_usage(const char *argv0) {
86   printf("usage: %s [--mmio-addr HEX] [--bridge-addr HEX] [--no-write-test] "
87          "[--scan]\n",
88          argv0);
89 }
90
91 int main(int argc, char **argv) {
92   off_t mmio_addr = k_default_vga_mmio_addr;
```

```
  off_t bridge_addr = k_default_bridge_reset_addr;
  int write_test = 1;
  int scan = 0;
  int mmio_addr_overridden = 0;
  int mem_fd;
  void *bridge_map = NULL;
  void *vga_map = NULL;
  size_t bridge_map_length = 0;
  size_t vga_map_length = 0;
  volatile uint32_t *bridge_reg = NULL;
  volatile uint32_t *vga_regs = NULL;
  uint32_t bridge_before;
  uint32_t bridge_after;
  uint32_t ident;
  int i;

  for (i = 1; i < argc; ++i) {
    if (strcmp(argv[i], "--mmio-addr") == 0 && i + 1 < argc) {
      if (parse_address(argv[++i], mmio_addr, &mmio_addr) != 0) {
        fprintf(stderr, "invalid --mmio-addr\n");
        return 1;
      }
      mmio_addr_overridden = 1;
    } else if (strcmp(argv[i], "--bridge-addr") == 0 && i + 1 < argc) {
      if (parse_address(argv[++i], bridge_addr, &bridge_addr) != 0) {
        fprintf(stderr, "invalid --bridge-addr\n");
        return 1;
      }
    } else if (strcmp(argv[i], "--no-write-test") == 0) {
      write_test = 0;
    } else if (strcmp(argv[i], "--scan") == 0) {
      scan = 1;
      write_test = 0;
    } else if (strcmp(argv[i], "--help") == 0) {
      print_usage(argv[0]);
      return 0;
    } else {
      fprintf(stderr, "unknown argument: %s\n", argv[i]);
      print_usage(argv[0]);
      return 1;
    }
  }

  (void)mmio_addr_overridden;

  mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (mem_fd < 0) {
    fprintf(stderr, "open /dev/mem failed: %s\n", strerror(errno));
    return 1;
  }

  if (map_physical(mem_fd, bridge_addr, sizeof(uint32_t), &bridge_map,
                   &bridge_map_length, &bridge_reg) != 0) {
    fprintf(stderr, "map bridge reset 0x%08lX failed: %s\n",
            (unsigned long)bridge_addr, strerror(errno));
    close(mem_fd);
    return 1;
  }

  if (map_physical(mem_fd, mmio_addr,
                   FIGHTER_VGA_MMIO_REG_SPAN_COUNT * sizeof(uint32_t),
                   &vga_map, &vga_map_length, &vga_regs) != 0) {
    fprintf(stderr, "map VGA MMIO 0x%08lX failed: %s\n",
            (unsigned long)mmio_addr, strerror(errno));
    unmap_region(&bridge_map, &bridge_map_length);
    close(mem_fd);
    return 1;
  }

  bridge_before = *bridge_reg;
  *bridge_reg = bridge_before & ~0x3U;
  bridge_after = *bridge_reg;
  ident = vga_regs[FIGHTER_VGA_MMIO_REG_IDENT];
```

```
166
167   printf("bridge_reset_addr : 0x%08lX\n", (unsigned long)bridge_addr);
168   printf("vga_mmio_addr     : 0x%08lX\n", (unsigned long)mmio_addr);
169   printf("bridge_before     : 0x%08" PRIX32 "\n", bridge_before);
170   printf("bridge_after      : 0x%08" PRIX32 "\n", bridge_after);
171   printf("ident             : 0x%08" PRIX32 " %s\n", ident,
172          ident == k_vga_ident ? "(VPGA OK)" : "(unexpected)");
173
174   if (scan) {
175     uint32_t word;
176
177     printf("scan              : reading 0x%08lX..0x%08lX\n",
178            (unsigned long)mmio_addr, (unsigned long)mmio_addr + 0x0fffUL);
179     for (word = 0; word < 0x1000U / sizeof(uint32_t); ++word) {
180       uint32_t value = vga_regs[word];
181       if (value == k_vga_ident) {
182         printf("found VPGA        : 0x%08" PRIX32 "\n",
183                (uint32_t)mmio_addr + word * 4U);
184       } else if (word < 64U) {
185         printf("word[0x%03" PRIX32 "]      : 0x%08" PRIX32 "\n",
186                word * 4U, value);
187       }
188     }
189   }
190
191   if (write_test) {
192     volatile uint32_t *frame =
193         vga_regs + FIGHTER_VGA_MMIO_REG_FRAME_WORD_OFFSET;
194     int x;
195     int y;
196
197     for (y = 0; y < FIGHTER_VGA_MMIO_FRAME_HEIGHT; ++y) {
198       for (x = 0; x < FIGHTER_VGA_MMIO_FRAME_WIDTH; x += 2) {
199         uint16_t p0 =
200             (uint16_t)((((unsigned int)x * 31U) /
201                         FIGHTER_VGA_MMIO_FRAME_WIDTH)
202                        << 11) |
203             (uint16_t)((((unsigned int)y * 63U) /
204                         FIGHTER_VGA_MMIO_FRAME_HEIGHT)
205                        << 5) |
206             0x000fU;
207         uint16_t p1 =
208             0xf800U |
209             (uint16_t)((((unsigned int)y * 63U) /
210                         FIGHTER_VGA_MMIO_FRAME_HEIGHT)
211                        << 5) |
212             (uint16_t)(((unsigned int)x * 31U) /
213                        FIGHTER_VGA_MMIO_FRAME_WIDTH);
214         frame[(y * FIGHTER_VGA_MMIO_FRAME_WIDTH + x) / 2] =
215             (uint32_t)p0 | ((uint32_t)p1 << 16);
216       }
217     }
218     vga_regs[FIGHTER_VGA_MMIO_REG_CONTROL] =
219         FIGHTER_VGA_MMIO_CONTROL_SWAP_REQUEST;
220     printf("write_test        : wrote RGB565 gradient frame (%dx%d)\n",
221            FIGHTER_VGA_MMIO_FRAME_WIDTH, FIGHTER_VGA_MMIO_FRAME_HEIGHT);
222   }
223
224   unmap_region(&vga_map, &vga_map_length);
225   unmap_region(&bridge_map, &bridge_map_length);
226   close(mem_fd);
227   return ident == k_vga_ident ? 0 : 2;
228 }
```

A.2 Software Header Files

A.2.1 sw/include/fighter_animation.h

```
1 #ifndef FIGHTER_ANIMATION_H
2 #define FIGHTER_ANIMATION_H
```

```
3
4  #include "fighter_game.h"
5
6  #ifdef __cplusplus
7  extern "C" {
8  #endif
9
10 #define FIGHTER_ANIMATION_MAX_PLAYERS FIGHTER_PLAYER_COUNT
11
12 typedef struct {
13   int width;
14   int height;
15   unsigned char *pixels;
16   char *source_path;
17 } fighter_sprite_t;
18
19 typedef struct {
20   fighter_sprite_t *frames;
21   int frame_count;
22   int ticks_per_frame;
23   int loop;
24 } fighter_animation_clip_t;
25
26 typedef struct {
27   fighter_animation_clip_t idle;
28   fighter_animation_clip_t walk;
29   fighter_animation_clip_t crouch;
30   fighter_animation_clip_t crouch_guard;
31   fighter_animation_clip_t jump;
32   fighter_animation_clip_t guard;
33   fighter_animation_clip_t block_stun;
34   fighter_animation_clip_t hit;
35   fighter_animation_clip_t ko;
36   fighter_animation_clip_t victory;
37
38   fighter_animation_clip_t normal_attack;
39   fighter_animation_clip_t fireball_attack;
40   fighter_animation_clip_t fireball_projectile;
41   fighter_animation_clip_t dragon_punch_attack;
42   fighter_animation_clip_t jump_attack;
43   fighter_animation_clip_t forward_jump_attack;
44   fighter_animation_clip_t back_jump_attack;
45   fighter_animation_clip_t sweep_attack;
46 } fighter_character_animation_set_t;
47
48 typedef struct {
49   const fighter_animation_clip_t *current_clip;
50   int frame_index;
51   int tick_in_frame;
52 } fighter_player_animation_state_t;
53
54 typedef struct {
55   fighter_character_animation_set_t ryu;
56   fighter_character_animation_set_t ken;
57   fighter_player_animation_state_t players[FIGHTER_ANIMATION_MAX_PLAYERS];
58 } fighter_animation_system_t;
59
60 int fighter_animation_system_init(fighter_animation_system_t *system);
61
62 int fighter_animation_system_init_with_roots(fighter_animation_system_t *system,
63                                              const char *ryu_root,
64                                              const char *ken_root);
65
66 void fighter_animation_system_close(fighter_animation_system_t *system);
67
68 void fighter_animation_system_update(fighter_animation_system_t *system,
69                                      const fighter_game_t *game);
70
71 const fighter_sprite_t *fighter_animation_current_sprite(
72     const fighter_animation_system_t *system,
73     int player_index);
74
75 const fighter_animation_clip_t *fighter_animation_current_clip(
```

```
    const fighter_animation_system_t *system,
    int player_index);

int fighter_animation_current_frame_index(
    const fighter_animation_system_t *system,
    int player_index);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.2 sw/include/fighter_audio.h

```
#ifndef FIGHTER_AUDIO_H
#define FIGHTER_AUDIO_H

#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef enum {
  FIGHTER_AUDIO_COMMAND_NONE = 0,
  FIGHTER_AUDIO_COMMAND_PLAY_ONCE,
  FIGHTER_AUDIO_COMMAND_START_LOOP,
  FIGHTER_AUDIO_COMMAND_STOP_LOOP
} fighter_audio_command_type_t;

typedef enum {
  FIGHTER_AUDIO_TRACK_NONE = 0,
  FIGHTER_AUDIO_TRACK_MENU_BGM,
  FIGHTER_AUDIO_TRACK_MENU_CONFIRM,
  FIGHTER_AUDIO_TRACK_GAME_OVER
} fighter_audio_track_t;

typedef struct {
  fighter_audio_command_type_t type;
  fighter_audio_track_t track;
} fighter_audio_command_t;

#define FIGHTER_AUDIO_MAX_COMMANDS 4

typedef struct {
  size_t count;
  fighter_audio_command_t commands[FIGHTER_AUDIO_MAX_COMMANDS];
} fighter_audio_command_list_t;

typedef enum {
  FIGHTER_AUDIO_BACKEND_DISABLED = 0,
  FIGHTER_AUDIO_BACKEND_COMMAND,
  FIGHTER_AUDIO_BACKEND_MMIO
} fighter_audio_backend_t;

typedef struct {
  int enable_command_audio;
} fighter_audio_options_t;

typedef struct {
  fighter_audio_backend_t backend;
  fighter_audio_track_t looping_track;
  int one_shot_logged;
  int backend_logged;
  int player_kind;
  int loop_pid;
  unsigned long mmio_addr;
  unsigned long bridge_reset_addr;
  char aplay_device[64];
```

```
  char status_detail[160];
  void *backend_data;
} fighter_audio_context_t;

void fighter_audio_command_list_clear(fighter_audio_command_list_t *list);
int fighter_audio_command_list_push(fighter_audio_command_list_t *list,
                                    fighter_audio_command_type_t type,
                                    fighter_audio_track_t track);

const char *fighter_audio_track_name(fighter_audio_track_t track);
const char *fighter_audio_track_path(fighter_audio_track_t track);
const char *fighter_audio_backend_name(const fighter_audio_context_t *context);

void fighter_audio_options_init(fighter_audio_options_t *options);
int fighter_audio_init(fighter_audio_context_t *context,
                       const fighter_audio_options_t *options);
void fighter_audio_close(fighter_audio_context_t *context);
void fighter_audio_process_commands(fighter_audio_context_t *context,
                                    const fighter_audio_command_list_t *commands);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.3 sw/include/fighter_game.h

```
#ifndef FIGHTER_GAME_H
#define FIGHTER_GAME_H

#include <stdint.h>

#include "fighter_audio.h"
#include "fighter_input.h"

#ifdef __cplusplus
extern "C" {
#endif

#define FIGHTER_PLAYER_COUNT 2

typedef enum {
  FIGHTER_GAME_STATE_MENU = 0,
  FIGHTER_GAME_STATE_PLAYING = 1,
  FIGHTER_GAME_STATE_GAME_OVER = 2
} fighter_game_state_t;

typedef enum {
  FIGHTER_CHARACTER_RYU = 0,
  FIGHTER_CHARACTER_KEN = 1
} fighter_character_id_t;

typedef enum {
  FIGHTER_VISUAL_STATE_IDLE = 0,
  FIGHTER_VISUAL_STATE_WALK,
  FIGHTER_VISUAL_STATE_JUMP,
  FIGHTER_VISUAL_STATE_CROUCH,
  FIGHTER_VISUAL_STATE_GUARD,
  FIGHTER_VISUAL_STATE_ATTACK,
  FIGHTER_VISUAL_STATE_HIT,
  FIGHTER_VISUAL_STATE_BLOCK_STUN,
  FIGHTER_VISUAL_STATE_KO,
  FIGHTER_VISUAL_STATE_VICTORY,
  FIGHTER_VISUAL_STATE_CROUCH_GUARD
} fighter_visual_state_t;

typedef enum {
  FIGHTER_ATTACK_PHASE_NONE = 0,
  FIGHTER_ATTACK_PHASE_STARTUP,
```

```
  FIGHTER_ATTACK_PHASE_ACTIVE,
  FIGHTER_ATTACK_PHASE_HIT_CONFIRM,
  FIGHTER_ATTACK_PHASE_BLOCK_CONFIRM,
  FIGHTER_ATTACK_PHASE_RECOVERY
} fighter_attack_phase_t;

typedef enum {
  FIGHTER_COMBAT_RESULT_NONE = 0,
  FIGHTER_COMBAT_RESULT_HIT,
  FIGHTER_COMBAT_RESULT_BLOCKED,
  FIGHTER_COMBAT_RESULT_TRADE,
  FIGHTER_COMBAT_RESULT_WHIFF
} fighter_combat_result_t;

typedef enum {
  FIGHTER_WINNER_NONE = 0,
  FIGHTER_WINNER_PLAYER1 = 1,
  FIGHTER_WINNER_PLAYER2 = 2,
  FIGHTER_WINNER_DRAW = 3
} fighter_winner_t;

typedef enum {
  FIGHTER_FINISH_REASON_NONE = 0,
  FIGHTER_FINISH_REASON_KO,
  FIGHTER_FINISH_REASON_TIME_OUT,
  FIGHTER_FINISH_REASON_DOUBLE_KO,
  FIGHTER_FINISH_REASON_EXIT
} fighter_finish_reason_t;

enum {
  FIGHTER_PLAYER_EVENT_NONE         = 0,
  FIGHTER_PLAYER_EVENT_ATTACK_START = 1 << 0,
  FIGHTER_PLAYER_EVENT_HIT          = 1 << 1,
  FIGHTER_PLAYER_EVENT_BLOCK        = 1 << 2,
  FIGHTER_PLAYER_EVENT_LAND         = 1 << 3,
  FIGHTER_PLAYER_EVENT_KO           = 1 << 4
};

typedef struct {
  int screen_width;
  int screen_height;
  int floor_y;
  int player_width;
  int player_height;
  int projectile_width;
  int projectile_height;
  int projectile_speed;
  int walk_speed;
  int jump_velocity;
  int gravity;
  int max_hp;
  int round_duration_frames;
  int menu_anim_period_frames;
  int game_over_anim_frames;
  int attack_cooldown_frames;
  int dragon_punch_lift_velocity;
  int attack_visual_frames;
  int hurt_visual_frames;
} fighter_game_config_t;

typedef struct {
  int x;
  int y;


  int vx;
  int vy;

  int hp;
  int facing;

  int attack_cooldown_frames;
  int attack_visual_frames;
```

```
  int hurt_visual_frames;
  int attack_phase_frames;
  int attack_has_connected;
  int block_stun_frames;

  fighter_attack_command_t last_attack;
  fighter_attack_phase_t attack_phase;
  fighter_combat_result_t combat_result;
  fighter_visual_state_t visual_state;
  fighter_character_id_t character_id;

  uint32_t event_flags;
  uint32_t state_frame;
} fighter_player_state_t;

typedef struct {
  int active;
  int owner_index;
  int x;
  int y;
  int vx;
  fighter_character_id_t character_id;
  uint32_t anim_ticks;
} fighter_projectile_state_t;

typedef struct {
  fighter_game_config_t config;
  fighter_game_state_t state;
  uint32_t frame_counter;
  uint32_t state_frames;
  uint32_t round_timer_frames;
  fighter_winner_t winner;
  fighter_finish_reason_t finish_reason;
  int menu_bgm_active;
  fighter_player_state_t players[FIGHTER_PLAYER_COUNT];
  fighter_projectile_state_t projectiles[FIGHTER_PLAYER_COUNT];
} fighter_game_t;

void fighter_game_config_default(fighter_game_config_t *config);
void fighter_game_init(fighter_game_t *game, const fighter_game_config_t *config);
void fighter_game_tick(fighter_game_t *game,
                       const fighter_player_result_t inputs[FIGHTER_PLAYER_COUNT],
                       fighter_audio_command_list_t *audio_commands);

int fighter_game_menu_animation_frame(const fighter_game_t *game);
int fighter_game_game_over_ready(const fighter_game_t *game);
int fighter_game_round_seconds_remaining(const fighter_game_t *game);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.4 sw/include/fighter_gamepad.h

```
#ifndef FIGHTER_GAMEPAD_H
#define FIGHTER_GAMEPAD_H
#include <stdbool.h>
#include "fighter_input.h"

#ifdef __cplusplus
extern "C" {
#endif


typedef struct {
  bool up;
  bool down;
  bool left;
```

```
  bool right;
  bool attack;
  bool guard;
  bool fireball;
  bool dragon_punch;
  bool start;
  bool exit_game;
} fighter_gamepad_buttons_t;


typedef struct {
  int fd;
  int connected;
  char device_path[128];
  fighter_gamepad_buttons_t current_buttons;
  fighter_gamepad_buttons_t previous_buttons;
} fighter_gamepad_t;


void fighter_gamepad_init(fighter_gamepad_t *gamepad);


int fighter_gamepad_open(fighter_gamepad_t *gamepad, const char *device_path);


void fighter_gamepad_close(fighter_gamepad_t *gamepad);


int fighter_gamepad_update(fighter_gamepad_t *gamepad,fighter_player_result_t *result);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.5 sw/include/fighter_input.h

```
#ifndef FIGHTER_INPUT_H
#define FIGHTER_INPUT_H

#include <stdbool.h>

#include "hid_keyboard_report.h"

#ifdef __cplusplus
extern "C" {
#endif

enum {
  FIGHTER_HID_KEY_A = 0x04,
  FIGHTER_HID_KEY_D = 0x07,
  FIGHTER_HID_KEY_J = 0x0d,
  FIGHTER_HID_KEY_K = 0x0e,
  FIGHTER_HID_KEY_L = 0x0f,
  FIGHTER_HID_KEY_S = 0x16,
  FIGHTER_HID_KEY_W = 0x1a
};

typedef enum {
  FIGHTER_MENU_ITEM_START = 0,
  FIGHTER_MENU_ITEM_EXIT = 1
} fighter_menu_item_t;

typedef enum {
  FIGHTER_MENU_ACTION_NONE = 0,
  FIGHTER_MENU_ACTION_MOVE_LEFT,
  FIGHTER_MENU_ACTION_MOVE_RIGHT,
  FIGHTER_MENU_ACTION_CONFIRM
} fighter_menu_action_t;
```

```
33
34 typedef enum {
35   FIGHTER_ATTACK_NONE = 0,
36   FIGHTER_ATTACK_NORMAL,
37   FIGHTER_ATTACK_FIREBALL,
38   FIGHTER_ATTACK_DRAGON_PUNCH,
39   FIGHTER_ATTACK_JUMP_ATTACK,
40   FIGHTER_ATTACK_FORWARD_JUMP_ATTACK,
41   FIGHTER_ATTACK_BACK_JUMP_ATTACK,
42   FIGHTER_ATTACK_SWEEP
43 } fighter_attack_command_t;
44
45 typedef struct {
46   bool up;
47   bool down;
48   bool left;
49   bool right;
50   bool attack;
51   bool guard;
52   bool exit_game;
53 } fighter_button_state_t;
54
55 typedef struct {
56   fighter_button_state_t previous_buttons;
57   fighter_menu_item_t selected_item;
58 } fighter_menu_parser_t;
59
60 typedef struct {
61   fighter_button_state_t previous_buttons;
62 } fighter_player_parser_t;
63
64 typedef struct {
65   fighter_menu_item_t selected_item;
66   fighter_menu_action_t action;
67 } fighter_menu_result_t;
68
69 typedef struct {
70   bool move_left;
71   bool move_right;
72   bool move_left_pressed;
73   bool move_right_pressed;
74   bool jump_held;
75   bool jump_pressed;
76   bool crouch_held;
77   bool crouch_pressed;
78   bool guard_held;
79   bool guard_pressed;
80   bool exit_requested;
81   bool attack_pressed;
82   bool any_input_active;
83   bool any_input_pressed;
84   fighter_attack_command_t attack_command;
85 } fighter_player_result_t;
86
87 void fighter_menu_parser_init(fighter_menu_parser_t *parser);
88 void fighter_player_parser_init(fighter_player_parser_t *parser);
89
90 void fighter_menu_parser_update(fighter_menu_parser_t *parser,
91                                 const usb_hid_keyboard_report_t *report,
92                                 fighter_menu_result_t *result);
93
94 void fighter_player_parser_update(fighter_player_parser_t *parser,
95                                   const usb_hid_keyboard_report_t *report,
96                                   fighter_player_result_t *result);
97
98 const char *fighter_attack_command_name(fighter_attack_command_t command);
99 const char *fighter_menu_item_name(fighter_menu_item_t item);
100
101 #ifdef __cplusplus
102 }
103 #endif
104
105 #endif
```

A.2.6 sw/include/fighter_mmio.h

```
#ifndef FIGHTER_MMIO_H
#define FIGHTER_MMIO_H

#include <stdint.h>

#include "fighter_game.h"

#ifdef __cplusplus
extern "C" {
#endif

enum {
  FIGHTER_MMIO_REG_GAME_STATE = 0,
  FIGHTER_MMIO_REG_PLAYER1_X,
  FIGHTER_MMIO_REG_PLAYER1_Y,
  FIGHTER_MMIO_REG_PLAYER1_STATE,
  FIGHTER_MMIO_REG_PLAYER2_X,
  FIGHTER_MMIO_REG_PLAYER2_Y,
  FIGHTER_MMIO_REG_PLAYER2_STATE,
  FIGHTER_MMIO_REG_PLAYER1_HP,
  FIGHTER_MMIO_REG_PLAYER2_HP,
  FIGHTER_MMIO_REG_ROUND_TIMER,
  FIGHTER_MMIO_REG_WINNER,
  FIGHTER_MMIO_REG_PLAYER1_FACING,
  FIGHTER_MMIO_REG_PLAYER2_FACING,
  FIGHTER_MMIO_REG_DEBUG_FLAGS,
  FIGHTER_MMIO_REG_PLAYER1_ATTACK_CMD,
  FIGHTER_MMIO_REG_PLAYER2_ATTACK_CMD,
  FIGHTER_MMIO_REG_PLAYER1_STATE_FRAME,
  FIGHTER_MMIO_REG_PLAYER2_STATE_FRAME,
  FIGHTER_MMIO_REG_PLAYER1_EVENT_FLAGS,
  FIGHTER_MMIO_REG_PLAYER2_EVENT_FLAGS,
  FIGHTER_MMIO_REG_PLAYER1_COMBAT_RESULT,
  FIGHTER_MMIO_REG_PLAYER2_COMBAT_RESULT,
  FIGHTER_MMIO_REG_COUNT
};

void fighter_mmio_encode(const fighter_game_t *game,
                         uint32_t regs[FIGHTER_MMIO_REG_COUNT]);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.7 sw/include/fighter_renderer.h

```
#ifndef FIGHTER_RENDERER_H
#define FIGHTER_RENDERER_H

#include <stdint.h>

#include "fighter_animation.h"
#include "fighter_game.h"

#ifdef __cplusplus
extern "C" {
#endif

typedef enum {
  FIGHTER_RENDERER_BACKEND_CONSOLE = 0,
  FIGHTER_RENDERER_BACKEND_FRAMEBUFFER = 1,
  FIGHTER_RENDERER_BACKEND_MMIO = 2
} fighter_renderer_backend_t;

typedef struct {
  int prefer_framebuffer;
```

```
  int console_interval_frames;
  const char *framebuffer_path;
} fighter_renderer_options_t;

typedef struct {
  int width;
  int height;
  unsigned char *pixels;
} fighter_rgb_image_t;

typedef struct {
  int width;
  int height;
  int stride;
  unsigned long data_length;
  unsigned char *pixels;
} fighter_fb_image_t;

typedef struct {
  fighter_renderer_backend_t backend;
  int console_interval_frames;
  char framebuffer_path_used[64];
  char init_status[256];

  uint32_t last_console_frame;
  fighter_game_state_t last_console_state;
  fighter_winner_t last_console_winner;
  fighter_finish_reason_t last_console_finish_reason;
  int last_console_ready;
  int last_console_valid;
  fighter_player_state_t last_console_players[FIGHTER_PLAYER_COUNT];

  int fb_fd;
  int fb_width;
  int fb_height;
  int fb_stride;
  int fb_bpp;
  unsigned char *fb_data;
  unsigned long fb_data_length;
  unsigned char *fb_backbuffer;
  unsigned long fb_backbuffer_length;

  fighter_rgb_image_t menu_frames[2];
  fighter_fb_image_t menu_frame_cache[2];
  fighter_rgb_image_t background_image;
  fighter_fb_image_t background_cache;

  int vga_mem_fd;
  void *vga_bridge_map;
  unsigned long vga_bridge_map_length;
  volatile uint32_t *vga_bridge_reset_reg;
  void *vga_regs_map;
  unsigned long vga_regs_map_length;
  volatile uint32_t *vga_regs;
  unsigned long vga_mmio_addr;
  unsigned long vga_bridge_reset_addr;
} fighter_renderer_t;

void fighter_renderer_options_init(fighter_renderer_options_t *options);

int fighter_renderer_init(fighter_renderer_t *renderer,
                          const fighter_renderer_options_t *options);

void fighter_renderer_close(fighter_renderer_t *renderer);

void fighter_renderer_draw(fighter_renderer_t *renderer,
                           const fighter_game_t *game,
                           const fighter_animation_system_t *anim_system);

const char *fighter_renderer_backend_name(const fighter_renderer_t *renderer);
const char *fighter_renderer_active_framebuffer_path(
    const fighter_renderer_t *renderer);
const char *fighter_renderer_status_detail(const fighter_renderer_t *renderer);
```

```
94
95 const char *fighter_renderer_menu_frame_path(int frame_index);
96
97 #ifdef __cplusplus
98 }
99 #endif
100
101 #endif
```

A.2.8 sw/include/fighter_ui.h

```
1  #ifndef FIGHTER_UI_H
2  #define FIGHTER_UI_H
3
4  #include "fighter_input.h"
5
6  typedef struct {
7    int bg_frame;
8    int frame_counter;
9    int blink_on;
10   int blink_counter;
11 } fighter_ui_context_t;
12
13 void fighter_ui_init(fighter_ui_context_t *ui);
14 void fighter_ui_update(fighter_ui_context_t *ui);
15
16
17 void fighter_ui_render_menu(const fighter_ui_context_t *ui,
18                             const fighter_menu_result_t *menu_result);
19
20 void fighter_ui_render_battle(const fighter_ui_context_t *ui);
21
22 #endif
```

A.2.9 sw/include/fighter_vga_mmio.h

```
1  #ifndef FIGHTER_VGA_MMIO_H
2  #define FIGHTER_VGA_MMIO_H
3
4  enum {
5    FIGHTER_VGA_MMIO_REG_CONTROL = 0,
6    FIGHTER_VGA_MMIO_REG_WIDTH = 1,
7    FIGHTER_VGA_MMIO_REG_HEIGHT = 2,
8    FIGHTER_VGA_MMIO_REG_STRIDE = 3,
9    FIGHTER_VGA_MMIO_REG_IDENT = 31,
10   FIGHTER_VGA_MMIO_REG_FRAME_WORD_OFFSET = 1024,
11
12   FIGHTER_VGA_MMIO_FRAME_WIDTH = 320,
13   FIGHTER_VGA_MMIO_FRAME_HEIGHT = 240,
14   FIGHTER_VGA_MMIO_FRAME_WORD_COUNT =
15       (FIGHTER_VGA_MMIO_FRAME_WIDTH * FIGHTER_VGA_MMIO_FRAME_HEIGHT) / 2,
16   FIGHTER_VGA_MMIO_REG_SPAN_COUNT =
17       FIGHTER_VGA_MMIO_REG_FRAME_WORD_OFFSET +
18       FIGHTER_VGA_MMIO_FRAME_WORD_COUNT
19 };
20
21 #define FIGHTER_VGA_MMIO_CONTROL_PRESENT (1U << 0)
22 #define FIGHTER_VGA_MMIO_CONTROL_SWAP_REQUEST (1U << 1)
23 #define FIGHTER_VGA_MMIO_CONTROL_SWAP_PENDING (1U << 1)
24 #define FIGHTER_VGA_MMIO_CONTROL_DISPLAY_BUFFER (1U << 8)
25 #define FIGHTER_VGA_MMIO_CONTROL_WRITE_BUFFER (1U << 9)
26
27 #endif
```

A.2.10 sw/include/hid_keyboard_report.h

```
#ifndef HID_KEYBOARD_REPORT_H
#define HID_KEYBOARD_REPORT_H

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
} usb_hid_keyboard_report_t;

void usb_hid_keyboard_report_clear(usb_hid_keyboard_report_t *report);
int usb_hid_keyboard_report_add_key(usb_hid_keyboard_report_t *report,
                                    uint8_t keycode);
int usb_hid_keyboard_report_contains(const usb_hid_keyboard_report_t *report,
                                     uint8_t keycode);
int usb_hid_keyboard_report_is_empty(const usb_hid_keyboard_report_t *report);

#ifdef __cplusplus
}
#endif

#endif
```

A.2.11 sw/include/usb_hid_keyboard.h

```
#ifndef USB_HID_KEYBOARD_H
#define USB_HID_KEYBOARD_H

#include <stddef.h>
#include <stdint.h>

#include "hid_keyboard_report.h"

#if defined(__has_include)
#  if __has_include(<libusb-1.0/libusb.h>)
#    include <libusb-1.0/libusb.h>
#  elif __has_include(<libusb.h>)
#    include <libusb.h>
#  else
#    error "libusb headers not found"
#  endif
#else
#  include <libusb-1.0/libusb.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif

#define USB_HID_KEYBOARD_MAX_DEVICES 2

typedef struct {
  libusb_device_handle *handle;
  uint8_t endpoint_address;
  int interface_number;
  int bus_number;
  int device_address;
  char product_name[128];
  usb_hid_keyboard_report_t last_report;
  int connected;
} usb_hid_keyboard_device_t;

typedef struct {
  libusb_context *context;
  usb_hid_keyboard_device_t devices[USB_HID_KEYBOARD_MAX_DEVICES];
```

```
  size_t device_count;
} usb_hid_keyboard_manager_t;

int usb_hid_keyboard_manager_init(usb_hid_keyboard_manager_t *manager,
                                  size_t max_devices);
void usb_hid_keyboard_manager_close(usb_hid_keyboard_manager_t *manager);

int usb_hid_keyboard_manager_poll(usb_hid_keyboard_manager_t *manager,
                                  usb_hid_keyboard_report_t *reports,
                                  size_t report_capacity,
                                  int timeout_ms);

#ifdef __cplusplus
}
#endif

#endif
```

A.3 Main Hardware SystemVerilog Files

A.3.1 hw/fighter_audio.sv

```
module fighter_audio_wm8731 #(
    parameter int FIFO_DEPTH = 128,
    parameter int CLK_HZ = 50000000,
    parameter int I2C_RATE_HZ = 100000,
    parameter int XCK_DIV = 4,
    parameter int BCLK_DIV = 16,
    parameter logic [6:0] I2C_DEVICE_ADDR = 7'h1A
) (
    input  logic        clk,
    input  logic        reset_n,

    input  logic        avs_chipselect,
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [1:0]  avs_address,
    input  logic [31:0] avs_writedata,
    output logic [31:0] avs_readdata,

    output logic        aud_xck,
    output logic        aud_bclk,
    output logic        aud_daclrck,
    output logic        aud_adclrck,
    output logic        aud_dacdat,
    input  logic        aud_adcdat,

    inout  wire         fpga_i2c_sclk,
    inout  wire         fpga_i2c_sdat,

    output logic        codec_init_done,
    output logic        codec_init_error
);

  /*
   * Register map expected by sw/audio/fighter_audio.c
   *
   * Word offset  Name       R/W  Description
   * 0            control    RW   write bit2/bit3 clears FIFOs, read returns status
   * 1            fifospace  R    [31:24]=left write space, [23:16]=right write space
   * 2            leftdata   W    left sample word; software writes int16 << 16
   * 3            rightdata  W    right sample word; software writes int16 << 16
   *
   * This module keeps the Avalon-MM side and the audio serializer in a single
   * clock domain driven by clk. With the default 50 MHz board clock:
   * - aud_xck  = clk / 4   = 12.5 MHz
   * - aud_bclk = clk / 16  = 3.125 MHz
   * - lrck     = clk / 1024 = 48.828125 kHz
   *
   * The codec is configured for left-justified, 16-bit samples, slave mode,
```

```
49    * DAC playback enabled, and active output. This is a practical bring-up
50    * implementation intended to line up with the current software stack.
51    */
52
53    localparam int FIFO_ADDR_WIDTH = (FIFO_DEPTH <= 1) ? 1 : $clog2(FIFO_DEPTH);
54    localparam int FIFO_COUNT_WIDTH = $clog2(FIFO_DEPTH + 1);
55    localparam int I2C_DIVIDER = ((CLK_HZ / (I2C_RATE_HZ * 4)) > 0) ?
56        (CLK_HZ / (I2C_RATE_HZ * 4)) : 1;
57    localparam int I2C_DIV_WIDTH = (I2C_DIVIDER <= 1) ? 1 : $clog2(I2C_DIVIDER);
58    localparam int XCK_HALF_DIV = (XCK_DIV / 2 > 0) ? (XCK_DIV / 2) : 1;
59    localparam int BCLK_HALF_DIV = (BCLK_DIV / 2 > 0) ? (BCLK_DIV / 2) : 1;
60    localparam int XCK_DIV_WIDTH = (XCK_HALF_DIV <= 1) ? 1 : $clog2(XCK_HALF_DIV);
61    localparam int BCLK_DIV_WIDTH = (BCLK_HALF_DIV <= 1) ? 1 : $clog2(BCLK_HALF_DIV);
62    localparam int INIT_LAST_INDEX = 10;
63
64    typedef enum logic [4:0] {
65      I2C_START_A,
66      I2C_START_B,
67      I2C_START_C,
68      I2C_LOAD_BYTE,
69      I2C_BIT_SETUP,
70      I2C_BIT_RISE,
71      I2C_BIT_FALL,
72      I2C_ACK_SETUP,
73      I2C_ACK_RISE,
74      I2C_ACK_SAMPLE,
75      I2C_STOP_A,
76      I2C_STOP_B,
77      I2C_STOP_C,
78      I2C_NEXT_WORD,
79      I2C_DONE,
80      I2C_ERROR
81    } i2c_state_t;
82
83    logic [31:0] left_fifo [0:FIFO_DEPTH-1];
84    logic [31:0] right_fifo[0:FIFO_DEPTH-1];
85
86    logic [FIFO_ADDR_WIDTH-1:0] left_wr_ptr;
87    logic [FIFO_ADDR_WIDTH-1:0] left_rd_ptr;
88    logic [FIFO_ADDR_WIDTH-1:0] right_wr_ptr;
89    logic [FIFO_ADDR_WIDTH-1:0] right_rd_ptr;
90    logic [FIFO_COUNT_WIDTH-1:0] left_count;
91    logic [FIFO_COUNT_WIDTH-1:0] right_count;
92
93    logic [63:0] shift_frame;
94    logic [5:0]  slot_bit_index;
95    logic [7:0]  left_space;
96    logic [7:0]  right_space;
97    logic [7:0]  left_count_status;
98    logic [7:0]  right_count_status;
99    logic        tx_underflow_seen;
100   logic        write_overflow_seen;
101
102   logic [XCK_DIV_WIDTH-1:0] xck_divider;
103   logic [BCLK_DIV_WIDTH-1:0] bclk_divider;
104
105   i2c_state_t i2c_state;
106   logic [I2C_DIV_WIDTH-1:0] i2c_divider;
107   logic [3:0] i2c_word_index;
108   logic [1:0] i2c_byte_index;
109   logic [2:0] i2c_bit_index;
110   logic [15:0] i2c_word;
111   logic [7:0] i2c_tx_byte;
112   logic i2c_tick;
113   logic i2c_scl_drive_low;
114   logic i2c_sdat_drive_low;
115   logic i2c_sdat_in;
116
117   function automatic logic [FIFO_ADDR_WIDTH-1:0] fifo_next_ptr(
118       input logic [FIFO_ADDR_WIDTH-1:0] ptr
119   );
120     if (ptr == FIFO_DEPTH - 1) begin
121       fifo_next_ptr = '0;
```

```
122     end else begin
123       fifo_next_ptr = ptr + 1'b1;
124     end
125   endfunction
126
127   function automatic logic [15:0] codec_init_word(
128       input logic [3:0] index
129   );
130     case (index)
131       4'd0: codec_init_word = 16'h1E00; // Reset.
132       4'd1: codec_init_word = 16'h0017; // Left line input mute.
133       4'd2: codec_init_word = 16'h0217; // Right line input mute.
134       4'd3: codec_init_word = 16'h0479; // Left output volume.
135       4'd4: codec_init_word = 16'h0679; // Right output volume.
136       4'd5: codec_init_word = 16'h0812; // DAC selected, bypass off.
137       4'd6: codec_init_word = 16'h0A00; // Digital path default.
138       4'd7: codec_init_word = 16'h0C00; // Power up everything.
139       4'd8: codec_init_word = 16'h0E01; // Left-justified, 16-bit, slave mode.
140       4'd9: codec_init_word = 16'h1000; // Normal mode, 256fs.
141       default: codec_init_word = 16'h1201; // Activate digital interface.
142     endcase
143   endfunction
144
145   function automatic logic [7:0] calc_space(
146       input logic [FIFO_COUNT_WIDTH-1:0] count
147   );
148     calc_space = FIFO_DEPTH - count;
149   endfunction
150
151   function automatic logic [7:0] codec_init_byte(
152       input logic [3:0] word_index,
153       input logic       high_byte
154   );
155     logic [15:0] word_value;
156     begin
157       word_value = codec_init_word(word_index);
158       if (high_byte) begin
159         codec_init_byte = word_value[15:8];
160       end else begin
161         codec_init_byte = word_value[7:0];
162       end
163     end
164   endfunction
165
166   assign left_space = calc_space(left_count);
167   assign right_space = calc_space(right_count);
168   assign left_count_status = left_count;
169   assign right_count_status = right_count;
170
171   assign fpga_i2c_sclk = i2c_scl_drive_low ? 1'b0 : 1'bz;
172   assign fpga_i2c_sdat = i2c_sdat_drive_low ? 1'b0 : 1'bz;
173   assign i2c_sdat_in = fpga_i2c_sdat;
174
175   always_comb begin
176     unique case (avs_address)
177       2'd0: avs_readdata = {
178           left_count_status,
179           right_count_status,
180           4'b0,
181           write_overflow_seen,
182           tx_underflow_seen,
183           codec_init_error,
184           codec_init_done
185       };
186       2'd1: avs_readdata = {left_space, right_space, 16'h0000};
187       // The sample write ports are write-only from software's point of view.
188       // Returning zero here avoids introducing asynchronous RAM reads that
189       // prevent the FIFOs from inferring to on-chip memory blocks.
190       2'd2: avs_readdata = 32'h00000000;
191       2'd3: avs_readdata = 32'h00000000;
192       default: avs_readdata = 32'h00000000;
193     endcase
194   end
```

```
195
196   always_ff @(posedge clk or negedge reset_n) begin : audio_core
197     logic clear_write_fifos;
198     logic clear_read_fifos;
199     logic left_push;
200     logic right_push;
201     logic left_pop;
202     logic right_pop;
203     logic [31:0] next_left_word;
204     logic [31:0] next_right_word;
205     logic bclk_falling_edge;
206
207     if (!reset_n) begin
208       left_wr_ptr <= '0;
209       left_rd_ptr <= '0;
210       right_wr_ptr <= '0;
211       right_rd_ptr <= '0;
212       left_count <= '0;
213       right_count <= '0;
214       shift_frame <= 64'h0000000000000000;
215       slot_bit_index <= 6'd0;
216       tx_underflow_seen <= 1'b0;
217       write_overflow_seen <= 1'b0;
218
219       aud_xck <= 1'b0;
220       aud_bclk <= 1'b0;
221       aud_daclrck <= 1'b0;
222       aud_adclrck <= 1'b0;
223       aud_dacdat <= 1'b0;
224       xck_divider <= '0;
225       bclk_divider <= '0;
226
227       i2c_divider <= '0;
228       i2c_state <= I2C_START_A;
229       i2c_word_index <= 4'd0;
230       i2c_byte_index <= 2'd0;
231       i2c_bit_index <= 3'd7;
232       i2c_word <= 16'h0000;
233       i2c_tx_byte <= 8'h00;
234       i2c_scl_drive_low <= 1'b0;
235       i2c_sdat_drive_low <= 1'b0;
236       codec_init_done <= 1'b0;
237       codec_init_error <= 1'b0;
238     end else begin
239       clear_write_fifos = avs_chipselect && avs_write && (avs_address == 2'd0) &&
240           avs_writedata[3];
241       clear_read_fifos = avs_chipselect && avs_write && (avs_address == 2'd0) &&
242           avs_writedata[2];
243       left_push = avs_chipselect && avs_write && (avs_address == 2'd2) &&
244           (left_count != FIFO_DEPTH);
245       right_push = avs_chipselect && avs_write && (avs_address == 2'd3) &&
246           (right_count != FIFO_DEPTH);
247       left_pop = 1'b0;
248       right_pop = 1'b0;
249       next_left_word = 32'h00000000;
250       next_right_word = 32'h00000000;
251       bclk_falling_edge = 1'b0;
252
253       if (avs_chipselect && avs_write && (avs_address == 2'd2) &&
254           (left_count == FIFO_DEPTH)) begin
255         write_overflow_seen <= 1'b1;
256       end
257       if (avs_chipselect && avs_write && (avs_address == 2'd3) &&
258           (right_count == FIFO_DEPTH)) begin
259         write_overflow_seen <= 1'b1;
260       end
261
262       if (clear_write_fifos || clear_read_fifos) begin
263         left_wr_ptr <= '0;
264         left_rd_ptr <= '0;
265         right_wr_ptr <= '0;
266         right_rd_ptr <= '0;
267         left_count <= '0;
```

```
268         right_count <= '0;
269         shift_frame <= 64'h0000000000000000;
270         slot_bit_index <= 6'd0;
271         tx_underflow_seen <= 1'b0;
272         write_overflow_seen <= 1'b0;
273       end else begin
274         if (left_push) begin
275           left_fifo[left_wr_ptr] <= avs_writedata;
276           left_wr_ptr <= fifo_next_ptr(left_wr_ptr);
277         end
278         if (right_push) begin
279           right_fifo[right_wr_ptr] <= avs_writedata;
280           right_wr_ptr <= fifo_next_ptr(right_wr_ptr);
281         end
282       end
283
284       if (xck_divider == XCK_HALF_DIV - 1) begin
285         xck_divider <= '0;
286         aud_xck <= ~aud_xck;
287       end else begin
288         xck_divider <= xck_divider + 1'b1;
289       end
290
291       if (bclk_divider == BCLK_HALF_DIV - 1) begin
292         bclk_divider <= '0;
293         bclk_falling_edge = aud_bclk;
294         aud_bclk <= ~aud_bclk;
295       end else begin
296         bclk_divider <= bclk_divider + 1'b1;
297       end
298
299       if (bclk_falling_edge) begin
300         aud_daclrck <= (slot_bit_index >= 6'd32);
301         aud_adclrck <= (slot_bit_index >= 6'd32);
302
303         if (slot_bit_index == 6'd0) begin
304           if ((left_count != 0) && (right_count != 0)) begin
305             next_left_word = left_fifo[left_rd_ptr];
306             next_right_word = right_fifo[right_rd_ptr];
307             left_pop = 1'b1;
308             right_pop = 1'b1;
309             shift_frame <= {next_left_word, next_right_word} << 1;
310             aud_dacdat <= next_left_word[31];
311           end else begin
312             tx_underflow_seen <= 1'b1;
313             shift_frame <= 64'h0000000000000000;
314             aud_dacdat <= 1'b0;
315           end
316         end else begin
317           aud_dacdat <= shift_frame[63];
318           shift_frame <= shift_frame << 1;
319         end
320
321         if (slot_bit_index == 6'd63) begin
322           slot_bit_index <= 6'd0;
323         end else begin
324           slot_bit_index <= slot_bit_index + 1'b1;
325         end
326       end
327
328       if (!clear_write_fifos && !clear_read_fifos) begin
329         unique case ({left_push, left_pop})
330           2'b10: left_count <= left_count + 1'b1;
331           2'b01: left_count <= left_count - 1'b1;
332           default: left_count <= left_count;
333         endcase
334
335         unique case ({right_push, right_pop})
336           2'b10: right_count <= right_count + 1'b1;
337           2'b01: right_count <= right_count - 1'b1;
338           default: right_count <= right_count;
339         endcase
340
```

```
341         if (left_pop) begin
342           left_rd_ptr <= fifo_next_ptr(left_rd_ptr);
343         end
344         if (right_pop) begin
345           right_rd_ptr <= fifo_next_ptr(right_rd_ptr);
346         end
347       end
348
349       if (i2c_divider == I2C_DIVIDER - 1) begin
350         i2c_divider <= '0;
351         i2c_tick = 1'b1;
352       end else begin
353         i2c_divider <= i2c_divider + 1'b1;
354         i2c_tick = 1'b0;
355       end
356
357       if (i2c_tick && !codec_init_done && !codec_init_error) begin
358         unique case (i2c_state)
359           I2C_START_A: begin
360             i2c_scl_drive_low <= 1'b0;
361             i2c_sdat_drive_low <= 1'b0;
362             i2c_state <= I2C_START_B;
363           end
364           I2C_START_B: begin
365             i2c_scl_drive_low <= 1'b0;
366             i2c_sdat_drive_low <= 1'b1;
367             i2c_state <= I2C_START_C;
368           end
369           I2C_START_C: begin
370             i2c_scl_drive_low <= 1'b1;
371             i2c_sdat_drive_low <= 1'b1;
372             i2c_state <= I2C_LOAD_BYTE;
373             i2c_byte_index <= 2'd0;
374           end
375           I2C_LOAD_BYTE: begin
376             i2c_word <= codec_init_word(i2c_word_index);
377             i2c_bit_index <= 3'd7;
378             unique case (i2c_byte_index)
379               2'd0: i2c_tx_byte <= {I2C_DEVICE_ADDR, 1'b0};
380               2'd1: i2c_tx_byte <= codec_init_byte(i2c_word_index, 1'b1);
381               default: i2c_tx_byte <= codec_init_byte(i2c_word_index, 1'b0);
382             endcase
383             i2c_state <= I2C_BIT_SETUP;
384           end
385           I2C_BIT_SETUP: begin
386             i2c_scl_drive_low <= 1'b1;
387             i2c_sdat_drive_low <= ~i2c_tx_byte[i2c_bit_index];
388             i2c_state <= I2C_BIT_RISE;
389           end
390           I2C_BIT_RISE: begin
391             i2c_scl_drive_low <= 1'b0;
392             i2c_state <= I2C_BIT_FALL;
393           end
394           I2C_BIT_FALL: begin
395             i2c_scl_drive_low <= 1'b1;
396             if (i2c_bit_index == 3'd0) begin
397               i2c_state <= I2C_ACK_SETUP;
398             end else begin
399               i2c_bit_index <= i2c_bit_index - 1'b1;
400               i2c_state <= I2C_BIT_SETUP;
401             end
402           end
403           I2C_ACK_SETUP: begin
404             i2c_scl_drive_low <= 1'b1;
405             i2c_sdat_drive_low <= 1'b0;
406             i2c_state <= I2C_ACK_RISE;
407           end
408           I2C_ACK_RISE: begin
409             i2c_scl_drive_low <= 1'b0;
410             i2c_state <= I2C_ACK_SAMPLE;
411           end
412           I2C_ACK_SAMPLE: begin
413             i2c_scl_drive_low <= 1'b1;
```

```
            if (i2c_sdat_in) begin
              codec_init_error <= 1'b1;
              i2c_state <= I2C_ERROR;
            end else if (i2c_byte_index == 2'd2) begin
              i2c_state <= I2C_STOP_A;
            end else begin
              i2c_byte_index <= i2c_byte_index + 1'b1;
              i2c_state <= I2C_LOAD_BYTE;
            end
          end
          I2C_STOP_A: begin
            i2c_scl_drive_low <= 1'b1;
            i2c_sdat_drive_low <= 1'b1;
            i2c_state <= I2C_STOP_B;
          end
          I2C_STOP_B: begin
            i2c_scl_drive_low <= 1'b0;
            i2c_sdat_drive_low <= 1'b1;
            i2c_state <= I2C_STOP_C;
          end
          I2C_STOP_C: begin
            i2c_scl_drive_low <= 1'b0;
            i2c_sdat_drive_low <= 1'b0;
            i2c_state <= I2C_NEXT_WORD;
          end
          I2C_NEXT_WORD: begin
            if (i2c_word_index == INIT_LAST_INDEX) begin
              codec_init_done <= 1'b1;
              i2c_state <= I2C_DONE;
            end else begin
              i2c_word_index <= i2c_word_index + 1'b1;
              i2c_state <= I2C_START_A;
            end
          end
          I2C_DONE: begin
            codec_init_done <= 1'b1;
          end
          I2C_ERROR: begin
            codec_init_error <= 1'b1;
          end
          default: begin
            i2c_state <= I2C_ERROR;
            codec_init_error <= 1'b1;
          end
        endcase
      end
    end
  end

endmodule
```

A.3.2 hw/fighter_vga_renderer.sv

```
module fighter_vga_renderer (
    input  logic        clk_50,
    input  logic        reset_n,

    input  logic        avs_chipselect,
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [15:0] avs_address,
    input  logic [31:0] avs_writedata,
    output logic [31:0] avs_readdata,

    output logic [7:0]  vga_r,
    output logic [7:0]  vga_g,
    output logic [7:0]  vga_b,
    output logic        vga_hs,
    output logic        vga_vs,
    output logic        vga_clk,
```

```
18      output logic        vga_blank_n,
19      output logic        vga_sync_n
20  );
21
22    localparam int H_VISIBLE = 640;
23    localparam int H_FRONT   = 16;
24    localparam int H_SYNC    = 96;
25    localparam int H_BACK    = 48;
26    localparam int H_TOTAL   = H_VISIBLE + H_FRONT + H_SYNC + H_BACK;
27
28    localparam int V_VISIBLE = 480;
29    localparam int V_FRONT   = 10;
30    localparam int V_SYNC    = 2;
31    localparam int V_BACK    = 33;
32    localparam int V_TOTAL   = V_VISIBLE + V_FRONT + V_SYNC + V_BACK;
33
34    localparam int FB_WIDTH       = 320;
35    localparam int FB_HEIGHT      = 240;
36    localparam int FB_WORDS_PER_ROW = FB_WIDTH / 2;
37    localparam int FB_WORD_COUNT  = FB_WORDS_PER_ROW * FB_HEIGHT;
38    localparam int FB_WORD_OFFSET = 1024;
39
40    localparam int REG_CONTROL = 0;
41    localparam int REG_WIDTH   = 1;
42    localparam int REG_HEIGHT  = 2;
43    localparam int REG_STRIDE  = 3;
44    localparam int REG_IDENT   = 31;
45
46    localparam logic [31:0] CONTROL_SWAP_REQUEST = 32'h00000002;
47    localparam logic [31:0] IDENT = 32'h56504741; // "VPGA"
48
49    logic        pixel_tick;
50    logic [9:0]  h_count;
51    logic [9:0]  v_count;
52    logic        visible;
53    logic        visible_d;
54    logic        pixel_half_d;
55    logic        display_buffer;
56    logic        swap_pending;
57    logic        swap_at_vblank;
58    logic        frame_write_enable;
59    logic        frame_write_buffer;
60    logic [15:0] frame_write_addr;
61    logic [15:0] frame_read_addr;
62    logic [31:0] read_word0;
63    logic [31:0] read_word1;
64    logic [31:0] read_word;
65    logic [15:0] pixel_rgb565;
66
67    assign vga_clk = pixel_tick;
68    assign vga_sync_n = 1'b0;
69    assign frame_write_enable =
70        avs_chipselect && avs_write && avs_address >= FB_WORD_OFFSET &&
71        avs_address < FB_WORD_OFFSET + FB_WORD_COUNT;
72    assign frame_write_addr = avs_address - FB_WORD_OFFSET;
73    assign frame_write_buffer = ~display_buffer;
74    assign swap_at_vblank = pixel_tick && h_count == 10'd0 && v_count == V_VISIBLE;
75    assign read_word = display_buffer ? read_word1 : read_word0;
76
77    function automatic logic [7:0] expand5(input logic [4:0] value);
78      expand5 = {value, value[4:2]};
79    endfunction
80
81    function automatic logic [7:0] expand6(input logic [5:0] value);
82      expand6 = {value, value[5:4]};
83    endfunction
84
85    altsyncram #(
86        .operation_mode("DUAL_PORT"),
87        .ram_block_type("M10K"),
88        .intended_device_family("Cyclone V"),
89        .width_a(32),
90        .widthad_a(16),
```

```
91        .numwords_a(FB_WORD_COUNT),
92        .width_b(32),
93        .widthad_b(16),
94        .numwords_b(FB_WORD_COUNT),
95        .width_byteena_a(1),
96        .outdata_reg_b("UNREGISTERED"),
97        .address_reg_b("CLOCK1"),
98        .read_during_write_mode_mixed_ports("OLD_DATA"),
99        .power_up_uninitialized("FALSE"),
100       .lpm_type("altsyncram")
101   ) framebuffer_ram0 (
102       .clock0(clk_50),
103       .clock1(clk_50),
104       .clocken0(1'b1),
105       .clocken1(1'b1),
106       .clocken2(1'b1),
107       .clocken3(1'b1),
108       .aclr0(1'b0),
109       .aclr1(1'b0),
110       .addressstall_a(1'b0),
111       .addressstall_b(1'b0),
112       .address_a(frame_write_addr),
113       .address_b(frame_read_addr),
114       .data_a(avs_writedata),
115       .data_b(32'h00000000),
116       .wren_a(frame_write_enable && !frame_write_buffer),
117       .wren_b(1'b0),
118       .rden_a(1'b1),
119       .rden_b(1'b1),
120       .byteena_a(1'b1),
121       .byteena_b(1'b1),
122       .q_a(),
123       .q_b(read_word0),
124       .eccstatus()
125   );
126
127   altsyncram #(
128       .operation_mode("DUAL_PORT"),
129       .ram_block_type("M10K"),
130       .intended_device_family("Cyclone V"),
131       .width_a(32),
132       .widthad_a(16),
133       .numwords_a(FB_WORD_COUNT),
134       .width_b(32),
135       .widthad_b(16),
136       .numwords_b(FB_WORD_COUNT),
137       .width_byteena_a(1),
138       .outdata_reg_b("UNREGISTERED"),
139       .address_reg_b("CLOCK1"),
140       .read_during_write_mode_mixed_ports("OLD_DATA"),
141       .power_up_uninitialized("FALSE"),
142       .lpm_type("altsyncram")
143   ) framebuffer_ram1 (
144       .clock0(clk_50),
145       .clock1(clk_50),
146       .clocken0(1'b1),
147       .clocken1(1'b1),
148       .clocken2(1'b1),
149       .clocken3(1'b1),
150       .aclr0(1'b0),
151       .aclr1(1'b0),
152       .addressstall_a(1'b0),
153       .addressstall_b(1'b0),
154       .address_a(frame_write_addr),
155       .address_b(frame_read_addr),
156       .data_a(avs_writedata),
157       .data_b(32'h00000000),
158       .wren_a(frame_write_enable && frame_write_buffer),
159       .wren_b(1'b0),
160       .rden_a(1'b1),
161       .rden_b(1'b1),
162       .byteena_a(1'b1),
163       .byteena_b(1'b1),
```

```
164       .q_a(),
165       .q_b(read_word1),
166       .eccstatus()
167   );
168
169   always_comb begin
170     if (avs_address == REG_IDENT) begin
171       avs_readdata = IDENT;
172     end else if (avs_address == REG_WIDTH) begin
173       avs_readdata = FB_WIDTH;
174     end else if (avs_address == REG_HEIGHT) begin
175       avs_readdata = FB_HEIGHT;
176     end else if (avs_address == REG_STRIDE) begin
177       avs_readdata = FB_WIDTH * 2;
178     end else if (avs_address == REG_CONTROL) begin
179       avs_readdata = 32'd1 |
180                      (swap_pending ? 32'h00000002 : 32'h00000000) |
181                      (display_buffer ? 32'h00000100 : 32'h00000000) |
182                      (frame_write_buffer ? 32'h00000200 : 32'h00000000);
183     end else begin
184       avs_readdata = 32'h00000000;
185     end
186   end
187
188   always_ff @(posedge clk_50 or negedge reset_n) begin
189     if (!reset_n) begin
190       display_buffer <= 1'b0;
191       swap_pending <= 1'b0;
192     end else begin
193       if (avs_chipselect && avs_write && avs_address == REG_CONTROL &&
194           (avs_writedata & CONTROL_SWAP_REQUEST) != 32'd0) begin
195         swap_pending <= 1'b1;
196       end
197
198       if (swap_pending && swap_at_vblank) begin
199         display_buffer <= ~display_buffer;
200         swap_pending <= 1'b0;
201       end
202     end
203   end
204
205   always_ff @(posedge clk_50 or negedge reset_n) begin
206     if (!reset_n) begin
207       pixel_tick <= 1'b0;
208     end else begin
209       pixel_tick <= ~pixel_tick;
210     end
211   end
212
213   always_ff @(posedge clk_50 or negedge reset_n) begin
214     if (!reset_n) begin
215       h_count <= 10'd0;
216       v_count <= 10'd0;
217     end else if (pixel_tick) begin
218       if (h_count == H_TOTAL - 1) begin
219         h_count <= 10'd0;
220         if (v_count == V_TOTAL - 1) begin
221           v_count <= 10'd0;
222         end else begin
223           v_count <= v_count + 10'd1;
224         end
225       end else begin
226         h_count <= h_count + 10'd1;
227       end
228     end
229   end
230
231   always_ff @(posedge clk_50 or negedge reset_n) begin
232     int source_x;
233     int source_y;
234     int read_index;
235
236     if (!reset_n) begin
```

```
237       frame_read_addr <= 16'd0;
238       visible_d <= 1'b0;
239       pixel_half_d <= 1'b0;
240     end else if (pixel_tick) begin
241       source_x = h_count[9:1];
242       source_y = v_count[8:1];
243       read_index = source_y * FB_WORDS_PER_ROW + source_x[8:1];
244
245       visible_d <= (h_count < H_VISIBLE) && (v_count < V_VISIBLE);
246       pixel_half_d <= source_x[0];
247       if (read_index >= 0 && read_index < FB_WORD_COUNT) begin
248         frame_read_addr <= read_index[15:0];
249       end else begin
250         frame_read_addr <= 16'd0;
251       end
252     end
253   end
254
255   always_comb begin
256     visible = (h_count < H_VISIBLE) && (v_count < V_VISIBLE);
257     vga_hs = ~((h_count >= H_VISIBLE + H_FRONT) &&
258                (h_count < H_VISIBLE + H_FRONT + H_SYNC));
259     vga_vs = ~((v_count >= V_VISIBLE + V_FRONT) &&
260                (v_count < V_VISIBLE + V_FRONT + V_SYNC));
261     vga_blank_n = visible;
262     pixel_rgb565 = pixel_half_d ? read_word[31:16] : read_word[15:0];
263   end
264
265   always_ff @(posedge clk_50 or negedge reset_n) begin
266     if (!reset_n) begin
267       vga_r <= 8'h00;
268       vga_g <= 8'h00;
269       vga_b <= 8'h00;
270     end else if (pixel_tick) begin
271       if (visible_d) begin
272         vga_r <= expand5(pixel_rgb565[15:11]);
273         vga_g <= expand6(pixel_rgb565[10:5]);
274         vga_b <= expand5(pixel_rgb565[4:0]);
275       end else begin
276         vga_r <= 8'h00;
277         vga_g <= 8'h00;
278         vga_b <= 8'h00;
279       end
280     end
281   end
282
283 endmodule
```

A.3.3 hw/soc_system_top.sv

```
1  // ====================================================================
2  // Copyright (c) 2013 by Terasic Technologies Inc.
3  // ====================================================================
4  //
5  // Modified 2019 by Stephen A. Edwards
6  //
7  // Permission:
8  //
9  //   Terasic grants permission to use and modify this code for use
10 //   in synthesis for all Terasic Development Boards and Altera
11 //   Development Kits made by Terasic.  Other use of this code,
12 //   including the selling ,duplication, or modification of any
13 //   portion is strictly prohibited.
14 //
15 // Disclaimer:
16 //
17 //   This VHDL/Verilog or C/C++ source code is intended as a design
18 //   reference which illustrates how these types of functions can be
19 //   implemented.  It is the user's responsibility to verify their
20 //   design for consistency and functionality through the use of
```

```
21 //   formal verification methods.  Terasic provides no warranty
22 //   regarding the use or functionality of this code.
23 //
24 // ============================================================
25 //
26 //  Terasic Technologies Inc
27 //
28 //  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
29 //
30 //                     web: http://www.terasic.com/
31 //                     email: support@terasic.com
32
33 module soc_system_top(
34
35  ///////// ADC /////////
36  inout         ADC_CS_N,
37  output        ADC_DIN,
38  input         ADC_DOUT,
39  output        ADC_SCLK,
40
41  ///////// AUD /////////
42  input         AUD_ADCDAT,
43  inout         AUD_ADCLRCK,
44  inout         AUD_BCLK,
45  output        AUD_DACDAT,
46  inout         AUD_DACLRCK,
47  output        AUD_XCK,
48
49  ///////// CLOCK2 /////////
50  input         CLOCK2_50,
51
52  ///////// CLOCK3 /////////
53  input         CLOCK3_50,
54
55  ///////// CLOCK4 /////////
56  input         CLOCK4_50,
57
58  ///////// CLOCK /////////
59  input         CLOCK_50,
60
61  ///////// DRAM /////////
62  output [12:0] DRAM_ADDR,
63  output [1:0]  DRAM_BA,
64  output        DRAM_CAS_N,
65  output        DRAM_CKE,
66  output        DRAM_CLK,
67  output        DRAM_CS_N,
68  inout  [15:0] DRAM_DQ,
69  output        DRAM_LDQM,
70  output        DRAM_RAS_N,
71  output        DRAM_UDQM,
72  output        DRAM_WE_N,
73
74  ///////// FAN /////////
75  output        FAN_CTRL,
76
77  ///////// FPGA /////////
78  output        FPGA_I2C_SCLK,
79  inout         FPGA_I2C_SDAT,
80
81  ///////// GPIO /////////
82  inout  [35:0] GPIO_0,
83  inout  [35:0] GPIO_1,
84
85  ///////// HEX0 /////////
86  output [6:0]  HEX0,
87
88  ///////// HEX1 /////////
89  output [6:0]  HEX1,
90
91  ///////// HEX2 /////////
92  output [6:0]  HEX2,
93
```

```
94  ///////// HEX3 /////////
95  output [6:0]  HEX3,
96
97  ///////// HEX4 /////////
98  output [6:0]  HEX4,
99
100 ///////// HEX5 /////////
101 output [6:0]  HEX5,
102
103 ///////// HPS /////////
104 inout         HPS_CONV_USB_N,
105 output [14:0] HPS_DDR3_ADDR,
106 output [2:0]  HPS_DDR3_BA,
107 output        HPS_DDR3_CAS_N,
108 output        HPS_DDR3_CKE,
109 output        HPS_DDR3_CK_N,
110 output        HPS_DDR3_CK_P,
111 output        HPS_DDR3_CS_N,
112 output [3:0]  HPS_DDR3_DM,
113 inout  [31:0] HPS_DDR3_DQ,
114 inout  [3:0]  HPS_DDR3_DQS_N,
115 inout  [3:0]  HPS_DDR3_DQS_P,
116 output        HPS_DDR3_ODT,
117 output        HPS_DDR3_RAS_N,
118 output        HPS_DDR3_RESET_N,
119 input         HPS_DDR3_RZQ,
120 output        HPS_DDR3_WE_N,
121 output        HPS_ENET_GTX_CLK,
122 inout         HPS_ENET_INT_N,
123 output        HPS_ENET_MDC,
124 inout         HPS_ENET_MDIO,
125 input         HPS_ENET_RX_CLK,
126 input  [3:0]  HPS_ENET_RX_DATA,
127 input         HPS_ENET_RX_DV,
128 output [3:0]  HPS_ENET_TX_DATA,
129 output        HPS_ENET_TX_EN,
130 inout         HPS_GSENSOR_INT,
131 inout         HPS_I2C1_SCLK,
132 inout         HPS_I2C1_SDAT,
133 inout         HPS_I2C2_SCLK,
134 inout         HPS_I2C2_SDAT,
135 inout         HPS_I2C_CONTROL,
136 inout         HPS_KEY,
137 inout         HPS_LED,
138 inout         HPS_LTC_GPIO,
139 output        HPS_SD_CLK,
140 inout         HPS_SD_CMD,
141 inout  [3:0]  HPS_SD_DATA,
142 output        HPS_SPIM_CLK,
143 input         HPS_SPIM_MISO,
144 output        HPS_SPIM_MOSI,
145 inout         HPS_SPIM_SS,
146 input         HPS_UART_RX,
147 output        HPS_UART_TX,
148 input         HPS_USB_CLKOUT,
149 inout  [7:0]  HPS_USB_DATA,
150 input         HPS_USB_DIR,
151 input         HPS_USB_NXT,
152 output        HPS_USB_STP,
153
154 ///////// IRDA /////////
155 input         IRDA_RXD,
156 output        IRDA_TXD,
157
158 ///////// KEY /////////
159 input  [3:0]  KEY,
160
161 ///////// LEDR /////////
162 output [9:0]  LEDR,
163
164 ///////// PS2 /////////
165 inout         PS2_CLK,
166 inout         PS2_CLK2,
```

```
167  inout         PS2_DAT,
168  inout         PS2_DAT2,
169
170  ///////// SW /////////
171  input  [9:0]  SW,
172
173  ///////// TD /////////
174  input         TD_CLK27,
175  input  [7:0]  TD_DATA,
176  input         TD_HS,
177  output        TD_RESET_N,
178  input         TD_VS,
179
180  ///////// VGA /////////
181  output [7:0]  VGA_B,
182  output        VGA_BLANK_N,
183  output        VGA_CLK,
184  output [7:0]  VGA_G,
185  output        VGA_HS,
186  output [7:0]  VGA_R,
187  output        VGA_SYNC_N,
188  output        VGA_VS
189 );
190
191    wire audio_init_done;
192    wire audio_init_error;
193    soc_system soc_system0(
194      .clk_clk                       ( CLOCK_50 ),
195      .reset_reset_n                 ( 1'b1 ),
196
197      .hps_ddr3_mem_a                ( HPS_DDR3_ADDR ),
198      .hps_ddr3_mem_ba               ( HPS_DDR3_BA ),
199      .hps_ddr3_mem_ck               ( HPS_DDR3_CK_P ),
200      .hps_ddr3_mem_ck_n             ( HPS_DDR3_CK_N ),
201      .hps_ddr3_mem_cke              ( HPS_DDR3_CKE ),
202      .hps_ddr3_mem_cs_n             ( HPS_DDR3_CS_N ),
203      .hps_ddr3_mem_ras_n            ( HPS_DDR3_RAS_N ),
204      .hps_ddr3_mem_cas_n            ( HPS_DDR3_CAS_N ),
205      .hps_ddr3_mem_we_n             ( HPS_DDR3_WE_N ),
206      .hps_ddr3_mem_reset_n          ( HPS_DDR3_RESET_N ),
207      .hps_ddr3_mem_dq               ( HPS_DDR3_DQ ),
208      .hps_ddr3_mem_dqs              ( HPS_DDR3_DQS_P ),
209      .hps_ddr3_mem_dqs_n            ( HPS_DDR3_DQS_N ),
210      .hps_ddr3_mem_odt              ( HPS_DDR3_ODT ),
211      .hps_ddr3_mem_dm               ( HPS_DDR3_DM ),
212      .hps_ddr3_oct_rzqin            ( HPS_DDR3_RZQ ),
213
214      .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
215      .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
216      .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
217      .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
218      .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
219      .hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
220      .hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO ),
221      .hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC ),
222      .hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
223      .hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
224      .hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
225      .hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1] ),
226      .hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2] ),
227      .hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3] ),
228
229      .hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD ),
230      .hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0] ),
231      .hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1] ),
232      .hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK ),
233      .hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2] ),
234      .hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3] ),
235
236      .hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0] ),
237      .hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1] ),
238      .hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2] ),
239      .hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3] ),
```

```
240       .hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4] ),
241       .hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5] ),
242       .hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6] ),
243       .hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7] ),
244       .hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT ),
245       .hps_hps_io_usb1_inst_STP     ( HPS_USB_STP ),
246       .hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR ),
247       .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT ),
248
249       .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK ),
250       .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
251       .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
252       .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS ),
253
254       .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX ),
255       .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX ),
256
257       .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT ),
258       .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK ),
259
260       .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT ),
261       .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK ),
262
263       .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
264       .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
265       .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),
266       .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
267       .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
268       .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
269       .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),
270
271       .fighter_audio_0_fighter_audio_audio_xck        ( AUD_XCK ),
272       .fighter_audio_0_fighter_audio_audio_bclk       ( AUD_BCLK ),
273       .fighter_audio_0_fighter_audio_audio_adclrck    ( AUD_ADCLRCK ),
274       .fighter_audio_0_fighter_audio_aud_daclrck      ( AUD_DACLRCK ),
275       .fighter_audio_0_fighter_audio_audio_dacdat     ( AUD_DACDAT ),
276       .fighter_audio_0_fighter_audio_audio_adcdat     ( AUD_ADCDAT ),
277       .fighter_audio_0_fighter_audio_audio_i2c_sclk   ( FPGA_I2C_SCLK ),
278       .fighter_audio_0_fighter_audio_audio_i2c_sdat   ( FPGA_I2C_SDAT ),
279       .fighter_audio_0_fighter_audio_audio_init_done  ( audio_init_done ),
280       .fighter_audio_0_fighter_audio_audio_init_error ( audio_init_error ),
281
282       .fighter_vga_0_vga_vga_r       ( VGA_R ),
283       .fighter_vga_0_vga_vga_g       ( VGA_G ),
284       .fighter_vga_0_vga_vga_b       ( VGA_B ),
285       .fighter_vga_0_vga_vga_hs      ( VGA_HS ),
286       .fighter_vga_0_vga_vga_vs      ( VGA_VS ),
287       .fighter_vga_0_vga_vga_clk     ( VGA_CLK ),
288       .fighter_vga_0_vga_vga_blank_n ( VGA_BLANK_N ),
289       .fighter_vga_0_vga_vga_sync_n  ( VGA_SYNC_N )
290     );
291
292     // The following quiet the "no driver" warnings for output
293     // pins and should be removed if you use any of these peripherals
294
295     assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
296     assign ADC_DIN  = SW[0];
297     assign ADC_SCLK = SW[0];
298
299     assign DRAM_ADDR = {13{SW[0]}};
300     assign DRAM_BA   = {2{SW[0]}};
301     assign DRAM_DQ   = SW[1] ? {16{SW[0]}} : {16{1'bZ}};
302     assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
303             DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = {8{SW[0]}};
304
305     assign FAN_CTRL = SW[0];
306
307     assign GPIO_0 = SW[1] ? {36{SW[0]}} : {36{1'bZ}};
308     assign GPIO_1 = SW[1] ? {36{SW[0]}} : {36{1'bZ}};
309
310     assign HEX0 = {7{SW[1]}};
311     assign HEX1 = {7{SW[2]}};
312     assign HEX2 = {7{SW[3]}};
```

```
313     assign HEX3 = {7{SW[4]}};
314     assign HEX4 = {7{SW[5]}};
315     assign HEX5 = {7{SW[6]}};
316 
317     assign IRDA_TXD = SW[0];
318 
319     assign LEDR = {8'b0, audio_init_error, audio_init_done};
320 
321     assign PS2_CLK  = SW[1] ? SW[0] : 1'bZ;
322     assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
323     assign PS2_DAT  = SW[1] ? SW[0] : 1'bZ;
324     assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;
325 
326     assign TD_RESET_N = SW[0];
327 
328 endmodule
```