XI ALESTE BIOS

ARCHITECTURE SPECIFICATION

h2w

9 января 2026 г.

Содержание

Status: Draft

1. DOCUMENT REVISION HISTORY

Version	Date	Changes	Author
1.1	2026-01-01	Simplified API, added memory context, fixed states	h2w
1.0	2024-01-15	Initial specification	h2w

2. PURPOSE

This document defines the architectural standards, interfaces, and requirements for device driver implementation in the Aleste BIOS system. It serves as the authoritative reference for driver developers and system integrators.

3. SCOPE

This specification covers: - Device driver architecture - Interfaces and protocols - Data formats and structures - Implementation requirements - Driver lifecycle and states

Excluded from this document are hardware-specific details, memory maps, and port addresses, which are defined in separate hardware specification documents.

4. ARCHITECTURAL OVERVIEW

4.1. Core Philosophy

The Aleste BIOS driver architecture follows the principle of <<speed through predictability.>> Unlike complex driver models that introduce abstraction layers at the cost of performance, this architecture prioritizes deterministic execution times and minimal overhead. Each architectural decision balances functionality with the constraints of 8-bit hardware.

The system achieves polymorphism without runtime indirection overhead by using fixed memory slots and copyable VTable structures. This approach maintains the simplicity of direct calls while enabling driver substitution when needed.

4.2. Performance Guarantees

Key performance characteristics: - Direct driver call: 17 clock cycles (JP + RET) - Context switch (when needed): ~30-50 cycles - Hot cache hit (same driver repeated): 0 extra cycles - Command execution: ≤ 100 cycles for common operations

These guarantees are achieved through fixed addressing, hash-based context validation, and minimal abstraction layers.

5. CORE PHILOSOPHY: SPEED THROUGH PREDICTABILITY

This specification embodies a fundamental principle: on 8-bit hardware, **abstraction costs clock cycles**. The Aleste BIOS driver architecture rejects complex runtime indirection models that add overhead for theoretical flexibility.

Our design is governed by Z80 and banked memory realities:

- **Every call matters**: At 50Hz frame rate, you have only 20ms per frame. Saving tens of cycles per operation determines whether you can have dynamic graphics or just static screens.
- **Memory is banked**: Switching memory contexts is expensive. We must minimize switches and make them lightning-fast when unavoidable.
- **Simplicity enables speed**: The fastest code never executes. The most predictable system is the simplest.

Thus, we achieve polymorphism not through function pointer tables (adding ~50 cycles) but through **physical copying of jump instructions** into fixed slots. We manage banked memory not with constant switching but with **hash-based lazy context validation**. The result:

- Direct driver call: exactly **17 clock cycles** (JP + RET)
- Context switch: **0 cycles** if already in correct bank

- Video driver switch: copying 15 bytes, with **zero runtime overhead** thereafter

This document specifies **how** to build for this reality—a blueprint for performance through simplicity.
The KISS Principle (Keep It Simple, Stupid)
A cardinal rule for all development under this specification:

> It is better NOT to declare and implement a function, variable, constant, or macro than to implement one without a clear, immediate purpose.

Complexity is the enemy of performance and reliability. Before adding any element:

- Is this required by the hardware?
- Does it eliminate a proven bottleneck?
- Does it simplify the driver API for the caller?

Do NOT create abstraction layers <<just in case.>> **Do NOT** add wrapper functions that only rename existing calls. **Do NOT** define constants for values used only once.

Every element must justify its existence by contributing directly to speed, predictability, or clarity.

6. DRIVER TYPES AND PATTERNS

6.1. Singleton Drivers

Singleton drivers represent hardware components that exist in a single, non-replaceable form within the system. These typically include core system components like memory management units, interrupt controllers, and system timers.

Singleton drivers feature fixed VTable implementations that never change during system operation. Their detection occurs once during system initialization, and they remain active for the duration of system operation. The system treats these drivers as fundamental building blocks that other drivers may depend upon.

6.2. Polymorphic Drivers

Polymorphic drivers support multiple implementations for a single hardware interface. Common examples include video controllers with different display modes, storage systems supporting various filesystem formats, and audio chips with alternative programming models.

These drivers employ a copy-on-activation mechanism where the system copies a specific implementation's VTable into a fixed memory slot. This approach enables runtime driver switching without modifying calling code or introducing indirect call overhead. All polymorphic drivers for a given interface share identical VTable layouts, ensuring compatibility.

7. DRIVER INTERFACES

7.1. Mandatory Driver Methods

Every driver, regardless of type or purpose, must implement exactly five core methods that form the foundation of driver management and operation.

detect() - Tests for the physical presence of the target device through minimal hardware verification. This method performs only basic communication tests without allocating resources. It must be safe to call at any time for hot-plug detection and returns success when the device responds predictably.

init() - Performs complete software initialization including memory allocation, data structure creation, and initial device configuration. This method prepares the driver for operational use but does not necessarily power on the physical device. Installation is idempotent, allowing safe multiple calls.

deinit() - Releases all software resources acquired during initialization and returns the driver to post-detect() state. This method must handle partial initialization states gracefully and ensure complete resource cleanup without affecting hardware power state.

get_info() - Returns immutable metadata about the driver implementation and device capabilities. The information block includes version data, capability flags, and descriptive strings that enable system-level driver management and dependency resolution.

command() - Provides unified access to all hardware control operations including power management, mode switching, and device-specific functionality. This extensible interface accepts standardized command codes with device-specific parameters, preventing VTable bloat while maintaining performance.

7.2. Driver Information Structure

The driver information structure provides standardized metadata that enables automated system management:

```
DriverInfo:
    .name_ptr:      dw 0        ; Pointer to ASCIIZ name string
    .version_major: db 0        ; Major version number
    .version_minor: db 0        ; Minor version number
    .flags:         dw 0        ; Capability flags
    .device_id:     db 0        ; Device type identifier
    .reserved:      db 0        ; Alignment padding
```

Capability flags include:

- `DRIVER_FLAG_HOTPLUG` (1 << 0): Supports hot-plug detection
- `DRIVER_FLAG_POWER_CTRL` (1 << 1): Supports power management commands
- `DRIVER_FLAG_HAS_IRQ` (1 << 2): Uses interrupt service routines
- `DRIVER_FLAG_POLYMORPHIC` (1 << 3): Multiple implementations available
- `DRIVER_FLAG_BANKED_MEM` (1 << 4): Requires banked memory context

8. DRIVER STATE MANAGEMENT

8.1. State Definitions

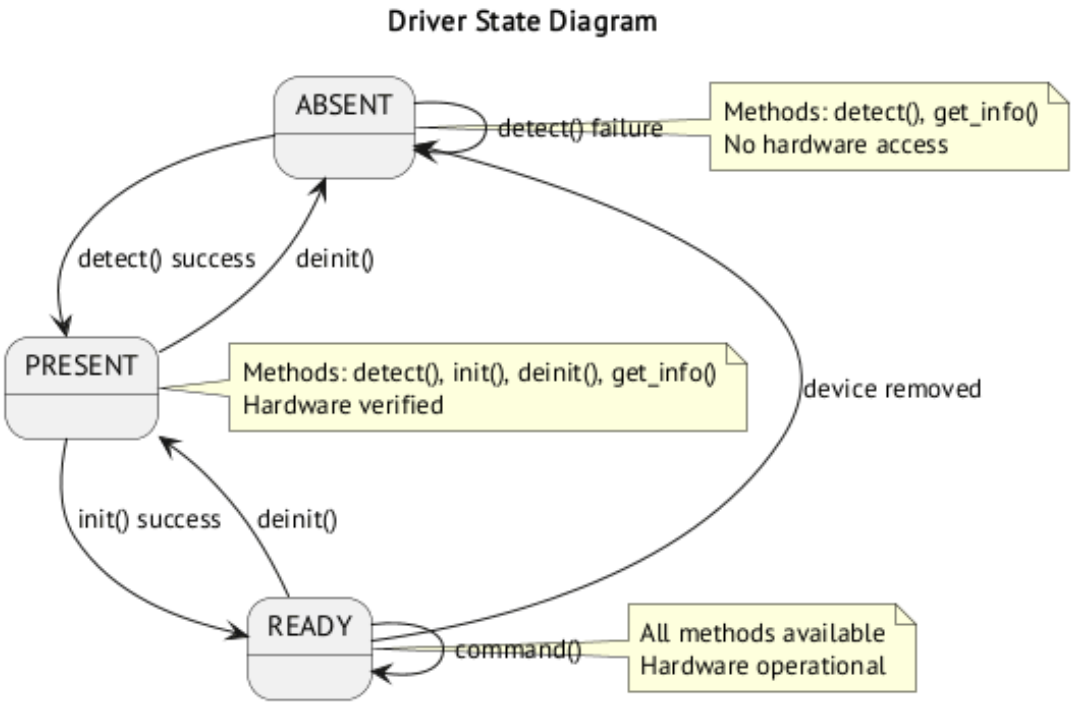
Drivers progress through three primary states with clear, unambiguous transitions:

ABSENT - The device is not detected or verified. The driver may be loaded in memory but cannot confirm hardware presence. Only the detect() and get_info() methods are valid in this state.

PRESENT - The device has been verified through successful detect() but software resources are not allocated. Hardware communication is established but the driver is not operational. The detect(), init(), deinit(), and get_info() methods are valid.

READY - The driver is fully initialized and operational. All API methods including device-specific functionality are available. The device may be in various power states (normal, suspended) controlled through the command() interface.

8.2. State Transition Diagram



8.3. State-Specific Method Validity

State	detect	init	deinit	command	get_info	Device Methods
ABSENT	+	-	-	-	+	-
PRESENT	+	+	+	+(1)	+	-
READY	+(2)	-	+	+	+	+

- (1) Only power management and status commands (POWER_ON, POWER_OFF, GET_STATUS)
- (2) Hot-plug verification only; successful detection does not change state from READY

9. MEMORY CONTEXT MANAGEMENT

9.1. Context Structure for Banked Memory

Drivers requiring specific memory bank configurations define MemoryContext structures that enable zero-overhead context switching:

```
;; MemoryContext Structure (8 bytes)
MemoryContext:
    .hash:      db 0    ;; XOR hash of bank configuration
    .slot_reg:  db 0    ;; Slot register value
    .bank_w0:   db 0    ;; Window 0 bank number
    .bank_w1:   db 0    ;; Window 1 bank number
    .bank_w2:   db 0    ;; Window 2 bank number
```

```
    .bank_w3:   db 0    ;; Window 3 bank number
    .flags:     db 0    ;; Context-specific flags
    .reserved:  db 0    ;; Alignment padding
```

9.2. Context Switching Protocol

Before executing driver code, the system ensures the correct memory context is active through a hash-based validation mechanism:

1. **Fast path**: Compare current context hash with driver's required hash
    - If hashes match: proceed immediately (5-8 cycle overhead)
    - If hashes differ: proceed to slow path
2. **Slow path**: Save current context, apply driver context, update cache
    - Save current bank configuration to previous_context buffer
    - Apply new configuration from driver's MemoryContext
    - Update current_context_hash and cache

This lazy switching approach eliminates context switch overhead for repeated calls to the same driver while maintaining correctness for cross-driver operations.

9.3. Context Caching (TLB)

In systems with banked memory architecture, drivers often require specific memory configurations to access their code and data. The Memory Context mechanism provides an efficient way to switch between these configurations with minimal overhead.

Each driver defines a MemoryContext structure that encapsulates its required memory configuration:

```
;; TLB Cache - 4 entries (32 bytes total)
context_tlb:
    hash:          db 0        ; Unique Hash value of the context
    slot_reg:      db 0        ; The content of the slot register
    mapper_state:  ds 4        ; The mapper's syaye
    ds 2                        ; 2 reserved bytes
```

The .hash field contains an XOR checksum of all configuration bytes (slot_reg ^ bank_w0 ^ bank_w1 ^ bank_w2 ^ bank_w3 ^ flags). This enables rapid comparison to determine if a context switch is necessary.

```
hash = slot_reg ^ bank_w0 ^ bank_w1 ^ bank_w2 ^ bank_w3 ^ flags
```

10. VTABLE ARCHITECTURE

10.1. VTable Structure and Layout

Driver VTable structures occupy fixed 64-byte memory slots and consist of contiguous JP (jump) instructions:

```
;; Standard VTable Layout (mandatory + optional methods)
__driver_vtable:
__driver_detect:      jp detect_impl      ;; Offset 0
__driver_init:        jp init_impl        ;; Offset 3
__driver_deinit:      jp deinit_impl      ;; Offset 6
__driver_get_info:    jp get_info_impl    ;; Offset 9
__driver_command:     jp command_impl     ;; Offset 12
__driver_method_1:    jp method_1_impl    ;; Offset 15 (optional)
__driver_method_2:    jp method_2_impl    ;; Offset 18 (optional)
;; ... up to 21 total methods (5 mandatory + 16 optional)
```

Each VTable slot provides space for 21 methods (64 bytes ÷ 3 bytes per JP). Unused method slots must point to a standard `_undefined_method` handler.

10.2. Polymorphic Driver Activation

Polymorphic drivers implement driver switching through the `CMD_SWITCH_DRIVER` command:

```
;; Example: Switching video driver implementation
switch_to_vga_mode:
    ld a, CMD_SWITCH_DRIVER
    ld hl, vga_driver_vtable
    call __video_command
    ret
```

Activation process: System validates new VTable size and structure 2. Current driver's context is saved (if in READY state) 3. New VTable is copied to the fixed memory slot 4. New driver's context is applied 5. System updates internal references

This mechanism provides zero-call-overhead polymorphism after activation while maintaining full driver state preservation.

11. COMMAND INTERFACE

11.1. Standard Command Set

The command() interface provides access to common device operations through standardized command codes:

Command	Code	Parameters	Description
POWER_ON	0x01	None	Enable device power
POWER_OFF	0x02	None	Disable device power
SUSPEND	0x03	None	Enter low-power suspend
RESUME	0x04	None	Resume from suspend
RESET	0x05	None	Hardware reset
GET_STATUS	0x06	None	Read device status
GET_CAPABILITIES	0x07	None	Read device capabilities
SET_CONFIG	0x08	HL=config_ptr	Apply configuration
GET_CONFIG	0x09	HL=buffer_ptr	Read configuration
SWITCH_DRIVER	0x10	HL=vtable_ptr	Activate new implementation

11.2. Device-Specific Command Ranges

Command codes are allocated in ranges to prevent conflicts:

- **0x00-0x0F**: System commands (common to all drivers)
- **0x10-0x1F**: Driver management commands
- **0x20-0x3F**: Audio device commands
- **0x40-0x5F**: Video device commands

- **0x60-0x7F**: Storage device commands
- **0x80-0x9F**: Network device commands
- **0xA0-0xFF**: Device-specific extensions

11.3. Power Management Example

```
;; Complete power management sequence
    call __sd_detect      ; Check device presence
    jr c, .no_device

    call __sd_init        ; Initialize software
    jr c, .init_failed

    ; Enable power (device-specific)
    ld a, CMD_POWER_ON
    call __sd_command

    ; Use device
    ld hl, buffer
    ld bc, sector_num
    call __sd_read_sector

    ; Suspend for power saving
    ld a, CMD_SUSPEND
    call __sd_command

    ; ... later, resume
    ld a, CMD_RESUME
    call __sd_command

    ; Clean shutdown
    call __sd_deinit
    ld a, CMD_POWER_OFF
    call __sd_command
```

12. ERROR HANDLING

12.1. Error Reporting Convention

All driver methods follow a uniform error reporting protocol:

- **Success**: Carry flag cleared (CF=0), register A = 0x00
- **Failure**: Carry flag set (CF=1), register A = error code
- **Additional data**: Registers may contain supplemental information

This enables efficient error checking with minimal instruction overhead:

```
    call __driver_method
    jr c, .handle_error
    ; Continue on success
```

12.2. Standard Error Codes

Code	Constant	Description
0x00	ERR_SUCCESS	Operation completed successfully
0x01	ERR_NOT_SUPPORTED	Operation not supported
0x02	ERR_NO_DEVICE	Target device not present
0x03	ERR_BAD_PARAMETER	Invalid parameter value
0x04	ERR_TIMEOUT	Operation timed out
0x05	ERR_BUSY	Device busy
0x06	ERR_NO_MEMORY	Insufficient memory
0x07	ERR_IO_ERROR	General I/O failure
0x08	ERR_WRONG_STATE	Invalid in current state
0x09	ERR_HARDWARE	Hardware failure
0x0A	ERR_CONFIG	Configuration error

13. DEPENDENCY MANAGEMENT

13.1. Dependency Declaration

Drivers declare dependencies through a null-terminated string list:

```
;; Example: Video driver dependencies
DEPENDS_ON:
    db "MMU", 0         ; Memory management unit
    db "PIC", 0         ; Programmable interrupt controller
    db 0                ; List terminator
```

13.2. Resolution Process

The system performs topological analysis during initialization, ensuring: 1. All prerequisites are satisfied before driver initialization 2. Circular dependencies are detected and reported 3. Missing dependencies prevent driver initialization with clear error reporting

14. PERFORMANCE REQUIREMENTS

14.1. Timing Constraints

- **detect()**: ≤ 1000 cycles (device presence verification)
- **init()/deinit()**: No specific limit (one-time operations)
- **command()**: ≤ 2000 cycles (control operations)
- **Context switch**: 0 cycles (cached), ≤ 50 cycles (uncached)
- **Interrupt handlers**: ≤ 200 cycles (minimal latency)

14.2. Memory Usage

- **VTable**: Fixed 64-byte slot
- **Driver code**: ≤ 4096 bytes recommended
- **Driver data**: ≤ 1024 bytes recommended

- **MemoryContext**: 8 bytes per required context
- **State information**: ≤ 256 bytes

15. COMPLIANCE AND TESTING

15.1. Compliance Verification

All drivers must pass the Driver Compliance Test Suite which validates: - API conformance and calling conventions - State transition correctness - Error handling protocols - Resource management (leak detection) - Performance requirements - Interoperability with system services

15.2. Quality Requirements

Production drivers must demonstrate: - 95%+ test coverage for all code paths - Stress testing under resource exhaustion - Fault injection testing for error recovery - Complete documentation of device-specific commands - Backward compatibility across minor versions

16. PRACTICAL IMPLEMENTATION GUIDANCE

16.1. Driver Patterns: Singleton vs. Polymorphic

The specification defines two driver types with distinct practical applications:

1. Singleton Drivers (Fixed Hardware)
    - **Examples**: System MMU, Primary Interrupt Controller (PIC), Core system timer
    - **Characteristics**: Represent unique, non-replaceable hardware. Their VTable is fixed at compile time.

- **Calling Convention**: Applications call their methods directly via fixed labels (e.g., `call __mmu_switch_bank`)
- **VTable Role**: Used internally by the system during boot—not a runtime polymorphism mechanism.

2. Polymorphic Drivers (Replaceable Interfaces)
   - **Examples**: Video controller (monochrome/VGA), Filesystem (FAT12/TR-DOS), Console output
   - **Characteristics**: Multiple implementations for a single abstract interface
   - **Calling Convention**: Applications **always** call the **polymorphic slot** (e.g., `call __video_draw_pixel`)
   - **Activation**: `CMD_SWITCH_DRIVER` copies the chosen VTable into the fixed slot—a **copy operation**, not pointer assignment
   - **Key Philosophy**: The application is **oblivious to implementation**
3. Practical Comparison

Таблица 1: Driver Patterns: Practical Differences

Aspect	Singleton Driver	Polymorphic Driver
Hardware	Unique, fixed device (MMU, CRTC)	Interface with multiple implementations
VTable in Memory	Fixed, immutable	Changes when implementation switches
Called by App	Directly via its own labels	Via polymorphic interface slot
Switching	Not applicable	Via `CMD_SWITCH_DRIVER`
Performance	Always 17-cycle direct call	17-cycle direct call (post-activation)

16.2. Concrete Conventions: Naming and Files

1. Naming Convention for Constants Constants follow a hierarchical pattern:

   `[DEVICE]_[TYPE]_[NAME]_[QUALIFIER]_[SUFFIX]`

Таблица 2: Constant Naming Components

Component	Description	Examples
DEVICE	Hardware component	`MMU_`, `VIDEO_`, `AUDIO_`, `SYS_`
TYPE	Entity kind	`PORT_` (I/O), `REG_` (MMIO), `CMD_`
NAME	Specific element	`CTRL`, `STATUS`, `DATA`, `MODE`
QUALIFIER	Clarifying modifier	`SUPER`, `USER`, `NATIVE`, `ENABLE`
SUFFIX	Value type	`_BIT`, `_MASK`, `_SIZE`, `_BASE`

Examples:

```
MMU_PORT_CTRL_SUPER_BIT    ; Single bit in MMU control port
VIDEO_CMD_SET_MODE         ; Command to set video mode
AUDIO_REG_VOLUME_A         ; MMIO register for audio volume
```

2. File Structure and Scope Rules
   - **One File, One Device**: Driver VTable, data, and code reside in a single `.asm` file
   - **Header Files (.inc) are for Declarations Only**:

  - Contain `extern` declarations
  - Contain `equ` or == for numeric constants
  - **Must NOT contain `defc`** (causes redefinition errors)
- **Implementation Files (.asm) are for Definitions**:
  - Use `defc` for linker symbols
  - Include relevant `.inc` files
- **C Interface Layer**: Separate `.h=/.c` pair marshals C stack arguments to Z80 registers

3. Project Directory Structure

```
XiAlesteBIOS/
├── include/                    # For C programs
│   ├── bios/                   # BIOS subsystems (C)
│   │   ├── drivers.h
│   │   └── errors.h
│   └── drivers/                # Public driver APIs (C)
│       ├── mmu.h
│       └── video.h
├── inc/                        # For Assembly
│   ├── bios/                   # BIOS subsystems (ASM)
│   │   ├── drivers.inc         # DRV_FLAG_*
│   │   └── errors.inc
│   ├── drivers/                # Driver interfaces (ASM)
│   │   ├── mmu.inc             # extern labels
│   │   └── video.inc
│   └── hardware/               # Hardware maps
│       ├── ports.inc
│       └── memory.inc
└── drivers/                    # IMPLEMENTATIONS
    ├── mmu/
    │   └── mmu.asm
    └── video/
        ├── video.asm           # Polymorphic slot
        └── video_mode0.asm     # Concrete implementation
```

16.3. Working with Real Hardware: MMU and Banked Memory

The `MemoryContext` structure (Section 4.1) applies practically to Aleste MMU:

1. Port Definitions (from `ports.inc`)

```
PORT_BANK_0        equ 0xFC    ; Window 0 (0000-3FFF)
PORT_BANK_1        equ 0xFD    ; Window 1 (4000-7FFF)
PORT_BANK_2        equ 0xFE    ; Window 2 (8000-BFFF)
PORT_BANK_3        equ 0xFF    ; Window 3 (C000-FFFF)
PORT_GLOBAL_CTRL   equ 0xF0    ; Master control register
```

2. Practical Macros (from `memory.inc`)

```
; Switch code bank (window 1). Preserves registers.
MACRO SWITCH_CODE_BANK bank
    ld a, bank
    out (PORT_BANK_1), a
ENDM

; Switch data bank (window 2).
MACRO SWITCH_DATA_BANK bank
    ld a, bank
    out (PORT_BANK_2), a
ENDM

; Get current memory slot (0-3).
MACRO GET_CURRENT_SLOT
    in a, (PORT_GLOBAL_CTRL)
    bit 0, a               ; Check SUPERVISOR bit
    jr z, .user_mode
    in a, (0xF9)           ; PORT_SUPER_SLOT
    jr .end
.user_mode:
    in a, (0xFB)           ; PORT_USER_SLOT
.end:
ENDM
```

3. Example: Driver with Banked Resources

```
; Driver context for sound with samples in specific bank
snd_context:
    .hash:      db 0xA5        ; Hash = 0x10 ^ 0x20 ^ 0x30
    .slot_reg:  db 1           ; Uses slot 1
    .bank_w0:   db 0x10        ; Window 0: System
    .bank_w1:   db 0x20        ; Window 1: Driver CODE
    .bank_w2:   db 0x30        ; Window 2: Sample DATA
    .bank_w3:   db 0x01        ; Window 3: System stack
    .flags:     db 0
    .reserved:  db 0

; Wrapper ensuring correct context
play_sound_safe:
    push hl
    ld hl, snd_context
    call ensure_memory_context   ; System routine (Section 4.2)
    pop hl
    call play_sound_impl        ; Actual implementation
    jp restore_previous_context
```

By adhering to these practical guidelines derived from hardware constraints, developers implement the formal specification while delivering the promised <<speed through predictability.>>

17. APPENDIX A: COMPLETE DRIVER TEMPLATE

17.1. Polymorphic Sound Driver

```
; ============================================================================
; AUDIO.ASM - Polymorphic Audio Driver Structure
; For Aleste LX BIOS
; ============================================================================

module audio_driver

include "../../inc/bios/undefined_method.inc"

include "../../inc/bios/drivers.inc"

; ----------------------------------------------------------------------------
; ПОЛИМОРФНАЯ VTABLE АУДИО
; ----------------------------------------------------------------------------

SECTION DRV_VTABLE
ALIGN 32

; Экспортируем ВСЕ метки VTable с двойным подчеркиванием
PUBLIC __audio_vtable
PUBLIC __audio_detect, __audio_install, __audio_command, __audio_uninstall
PUBLIC __audio_play_note, __audio_stop, __audio_set_volume
PUBLIC __audio_set_noise_type, __audio_set_noise_custom, __audio_enable_noise
PUBLIC __audio_set_envelope, __audio_enable_envelope, __audio_enable_tone
PUBLIC __audio_read_register, __audio_write_register
PUBLIC __audio_play_chord, __audio_beep

__audio_vtable:
__audio_detect:           jp _undefined_method    ; 0
__audio_install:          jp _undefined_method    ; 3
__audio_command:          jp _undefined_method    ; 6
__audio_uninstall:        jp _undefined_method    ; 9
__audio_play_note:        jp _undefined_method    ; 12
__audio_stop:             jp _undefined_method    ; 15
__audio_set_volume:       jp _undefined_method    ; 18
__audio_set_noise_type:   jp _undefined_method    ; 21
__audio_set_noise_custom: jp _undefined_method    ; 24
__audio_enable_noise:     jp _undefined_method    ; 27
__audio_set_envelope:     jp _undefined_method    ; 30
__audio_enable_envelope:  jp _undefined_method    ; 33
__audio_enable_tone:      jp _undefined_method    ; 36
__audio_read_register:    jp _undefined_method    ; 39
__audio_write_register:   jp _undefined_method    ; 42
__audio_play_chord:       jp _undefined_method    ; 45
__audio_beep:             jp _undefined_method    ; 48
__audio_vtable_end:

; ----------------------------------------------------------------------------
; ЗАВИСИМОСТИ И ДАННЫЕ
```

```
; ------------------------------------------------------------------------

SECTION DRV_DATA
DEPENDS_ON:
    db 0  ; Конец списка


_audio_info:
    dw @name
    db 1, 0        ; Версия 1.0
    dw API_STATIC_BIT      ; Флаги: DRIVER_FLAG_STATIC
    db 0           ; Нет IRQ
    db 0           ; Резерв
@name:
    db "AY-3-8910 Audio Driver", 0


SECTION DRV_CODE

; ------------------------------------------------------------------------
; Системные функции управления полиморфизмом
; ------------------------------------------------------------------------

; Активировать конкретный драйвер палитры
; Вход: HL = адрес VTable конкретного драйвера
PUBLIC __audio_activate_driver
__audio_activate_driver:
    ld de, __audio_vtable
    ld (current_driver), de
    ld bc, __audio_vtable_end-__audio_vtable
    ldir
    ret

; Получить текущую активную VTable
; Выход: HL = адрес текущей VTable
PUBLIC __audio_get_current_vtable
__audio_get_current_vtable:
    ld hl, (current_driver)
    ret


SECTION DRV_DATA
current_driver: dw 0x0000
```

17.2. AY8910 Sound Driver

```
; ========================================================================
; AUDIO_AY8910.ASM - AY-3-8910 Specific Driver
; Конкретная реализация семантического API
; ========================================================================

module audio_ay8910
```

```
include "../../inc/bios/undefined_method.inc"

; ------------------------------------------------------------------------------
; ВСЕ defc КОНСТАНТЫ ЭТОГО ДРАЙВЕРА
; ------------------------------------------------------------------------------

; Порты AY
defc AY_REGISTER_PORT = 0xFD
defc AY_DATA_PORT = 0xFE

; Регистры AY
defc AY_REG_TONE_A_FINE = 0
defc AY_REG_TONE_A_COARSE = 1
defc AY_REG_TONE_B_FINE = 2
defc AY_REG_TONE_B_COARSE = 3
defc AY_REG_TONE_C_FINE = 4
defc AY_REG_TONE_C_COARSE = 5
defc AY_REG_NOISE_PERIOD = 6
defc AY_REG_ENABLE = 7
defc AY_REG_VOLUME_A = 8
defc AY_REG_VOLUME_B = 9
defc AY_REG_VOLUME_C = 10
defc AY_REG_ENV_FINE = 11
defc AY_REG_ENV_COARSE = 12
defc AY_REG_ENV_SHAPE = 13
defc AY_REG_IO_A = 14
defc AY_REG_IO_B = 15

; Предустановки шума
defc NOISE_PERIOD_LOW = 0x1F      ; ~30 Гц
defc NOISE_PERIOD_MEDIUM = 0x0F   ; ~500 Гц
defc NOISE_PERIOD_HIGH = 0x03     ; ~15 кГц

; Предустановки огибающей
defc ENV_PERIOD_SLOW = 0x0FFF
defc ENV_PERIOD_MEDIUM = 0x07FF
defc ENV_PERIOD_FAST = 0x01FF
defc ENV_PERIOD_VERY_FAST = 0x007F

; Формы огибающей
defc ENV_SHAPE_DOWN = 0x00
defc ENV_SHAPE_UP = 0x0C
defc ENV_SHAPE_TRIANGLE = 0x0A
defc ENV_SHAPE_SAWTOOTH = 0x08

; Громкость
defc VOLUME_NORMAL = 15

; ------------------------------------------------------------------------------
; VTABLE КОНКРЕТНОГО ДРАЙВЕРА AY-3-8910
; ------------------------------------------------------------------------------
```

```
SECTION DRV_VTABLE
ALIGN 32

; Экспортируем VTable драйвера
PUBLIC __audio_ay8910_vtable

__audio_ay8910_vtable:
    jp _audio_ay8910_detect
    jp _audio_ay8910_install
    jp _audio_ay8910_command
    jp _audio_ay8910_uninstall
    jp _audio_ay8910_play_note
    jp _audio_ay8910_stop
    jp _audio_ay8910_set_volume
    jp _audio_ay8910_set_noise_type
    jp _audio_ay8910_set_noise_custom
    jp _audio_ay8910_enable_noise
    jp _audio_ay8910_set_envelope
    jp _audio_ay8910_enable_envelope
    jp _audio_ay8910_enable_tone
    jp _audio_ay8910_read_register
    jp _audio_ay8910_write_register
    jp _audio_ay8910_play_chord
    jp _audio_ay8910_beep

; ------------------------------------------------------------------------------
; ДАННЫЕ ДРАЙВЕРА
; ------------------------------------------------------------------------------

SECTION DRV_DATA

; Тень регистров (16 байт)
_ay_shadow_regs:
    ds 16

; Текущий выбранный регистр
_ay_selected_reg:
    db 0

; Флаги инициализации
_ay_initialized:
    db 0

; Предустановленные периоды шума
_noise_periods:
    db 0            ; NOISE_OFF
    db NOISE_PERIOD_LOW
    db NOISE_PERIOD_MEDIUM
    db NOISE_PERIOD_HIGH
    db 0            ; NOISE_CUSTOM будет( установлен отдельно)

; Предустановленные периоды огибающей
_env_periods:
```

```
    dw 0            ; ENVELOPE_OFF
    dw ENV_PERIOD_SLOW
    dw ENV_PERIOD_MEDIUM
    dw ENV_PERIOD_FAST
    dw ENV_PERIOD_VERY_FAST

; Формы огибающей
_env_shapes:
    db 0            ; ENVELOPE_OFF
    db ENV_SHAPE_DOWN
    db ENV_SHAPE_UP
    db ENV_SHAPE_TRIANGLE
    db ENV_SHAPE_SAWTOOTH

; Таблица частот нот (96 нот × 2 байта)
_notes_freq_table:
    dw 3822, 3608, 3405, 3214, 3034, 2863, 2703, 2551, 2408, 2273, 2145, 2025
    dw 1911, 1804, 1703, 1607, 1517, 1432, 1351, 1276, 1204, 1136, 1073, 1012
    dw 956, 902, 851, 804, 758, 716, 676, 638, 602, 568, 536, 506
    dw 478, 451, 426, 402, 379, 358, 338, 319, 301, 284, 268, 253
    dw 239, 225, 213, 201, 190, 179, 169, 159, 150, 142, 134, 127
    dw 119, 113, 106, 100, 95, 89, 84, 80, 75, 71, 67, 63
    dw 60, 56, 53, 50, 47, 45, 42, 40, 38, 36, 34, 32
    dw 30, 28, 26, 25, 24, 22, 21, 20, 19, 18, 17, 16

; ------------------------------------------------------------------------------
; ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ приватные(, без подчеркиваний)
; ------------------------------------------------------------------------------

SECTION DRV_CODE

; Выбрать регистр AY
; Вход: A = номер регистра
ay_select_register:
    push bc
    ld (_ay_selected_reg), a
    ld c, AY_REGISTER_PORT
    out (c), a
    pop bc
    ret

; Записать в выбранный регистр
; Вход: A = значение
ay_write_register:
    push bc
    push hl

    ; Обновляем тень
    ld hl, _ay_shadow_regs
    ld c, a                     ; Сохраняем значение
    ld a, (_ay_selected_reg)
    ld b, 0
    add hl, bc
```

```
    ld (hl), a                  ; Записываем в тень

    ; Пишем в чип
    ld a, c                     ; Восстанавливаем значение
    ld c, AY_DATA_PORT
    out (c), a

    pop hl
    pop bc
    ret

; Прочитать из выбранного регистра из( тени)
; Выход: A = значение
ay_read_shadow:
    push hl
    push bc
    ld hl, _ay_shadow_regs
    ld a, (_ay_selected_reg)
    ld c, a
    ld b, 0
    add hl, bc
    ld a, (hl)
    pop bc
    pop hl
    ret

; Получить частоту ноты
; Вход: A = номер ноты (0-95)
; Выход: DE = частота младший(, старший)
_get_note_frequency:
    push hl
    push bc

    ; Проверяем NOTE_SILENCE
    cp 255
    jr z, @silence

    ; Получаем адрес в таблице частот
    ld hl, _notes_freq_table
    add a, a           ; Умножаем на 2
    ld c, a
    ld b, 0
    add hl, bc

    ; Загружаем частоту
    ld e, (hl)
    inc hl
    ld d, (hl)
    jr @exit

@silence:
    ; Для тишины возвращаем 0
    ld de, 0
```

```
@exit:
    pop bc
    pop hl
    ret

; Получить регистр громкости для канала
; Вход: A = канал (0-2)
; Выход: A = номер регистра громкости (8-10)
_get_volume_register:
    add a, AY_REG_VOLUME_A
    ret

; Получить бит маски для канала в регистре 7
; Вход: A = канал (0-2)
; Выход: C = бит для тона, B = бит для шума
_get_channel_bits:
    ld c, 1           ; Бит 0 для канала A
    ld b, 8           ; Бит 3 для шума канала A

    cp 0
    ret z

    ld c, 2           ; Бит 1 для канала B
    ld b, 16          ; Бит 4 для шума канала B

    cp 1
    ret z

    ld c, 4           ; Бит 2 для канала C
    ld b, 32          ; Бит 5 для шума канала C
    ret

; ------------------------------------------------------------------------------
; ПУБЛИЧНЫЕ МЕТОДЫ c( двойным подчеркиванием)
; ------------------------------------------------------------------------------

PUBLIC _audio_ay8910_detect
_audio_ay8910_detect:
    push bc
    push de
    push hl

    ; Тест записичтения/ регистра
    ld a, AY_REG_VOLUME_A
    call ay_select_register
    ld a, 0x0A
    call ay_write_register

    ; Небольшая задержка
    ld bc, 100
@delay:
    dec bc
```

```
    ld a, b
    or c
    jr nz, @delay

    ; Проверяем
    call ay_read_shadow
    cp 0x0A
    jr nz, @not_found

    ; Успех
    xor a
    jr @exit

@not_found:
    ld a, 2  ; AUDIO_ERR_HARDWARE_ERROR
    scf

@exit:
    pop hl
    pop de
    pop bc
    ret

PUBLIC _audio_ay8910_install
_audio_ay8910_install:
    push af
    push bc
    push hl

    ld a, (_ay_initialized)
    or a
    jr nz, @already_init

    ; Сбрасываем тень регистров
    ld hl, _ay_shadow_regs
    ld de, _ay_shadow_regs + 1
    ld bc, 15
    ld (hl), 0
    ldir

    ; Сбрасываем все регистры чипа
    ld b, 16
    ld c, 0
@reset_loop:
    ld a, c
    call ay_select_register
    xor a
    call ay_write_register
    inc c
    djnz @reset_loop

    ; Базовая настройка
    ld a, 0x07        ; Включить все тоны
```

```
    ld c, AY_REG_ENABLE
    call ay_select_register
    call ay_write_register

    ; Установить громкость по умолчанию
    ld a, VOLUME_NORMAL
    ld c, AY_REG_VOLUME_A
    call ay_select_register
    call ay_write_register
    ld c, AY_REG_VOLUME_B
    call ay_select_register
    call ay_write_register
    ld c, AY_REG_VOLUME_C
    call ay_select_register
    call ay_write_register

    ; Флаг инициализации
    ld a, 1
    ld (_ay_initialized), a

@already_init:
    xor a  ; Успех

    pop hl
    pop bc
    pop af
    ret

PUBLIC _audio_ay8910_command
_audio_ay8910_command:
    ; TODO: реализовать команды
    ld a, 1  ; ERR_NOT_SUPPORTED
    scf
    ret

PUBLIC _audio_ay8910_uninstall
_audio_ay8910_uninstall:
    call _audio_ay8910_stop
    xor a
    ld (_ay_initialized), a
    ret

PUBLIC _audio_ay8910_play_note
_audio_ay8910_play_note:
    ; A = нота, B = канал, C = громкость
    push af
    push bc
    push de
    push hl

    ; Проверяем канал
    ld a, b
    cp 3
```

```
    jr nc, @invalid_channel

    ; Получаем частоту ноты
    pop af           ; Восстанавливаем A ноту()
    push af
    call _get_note_frequency
    ; DE содержит частоту

    ; Определяем регистры для канала
    pop af           ; Восстанавливаем регистры
    pop bc
    push bc
    push af

    ld a, b          ; Канал
    cp 0
    jr z, @ch_a
    cp 1
    jr z, @ch_b
    ; Канал C

@ch_c:
    ; Частота регистры( 4-5)
    ld a, e
    ld c, AY_REG_TONE_C_FINE
    call ay_select_register
    call ay_write_register
    ld a, d
    ld c, AY_REG_TONE_C_COARSE
    call ay_select_register
    call ay_write_register

    ; Громкость регистр( 10)
    ld a, 10
    jr @set_volume

@ch_b:
    ; Частота регистры( 2-3)
    ld a, e
    ld c, AY_REG_TONE_B_FINE
    call ay_select_register
    call ay_write_register
    ld a, d
    ld c, AY_REG_TONE_B_COARSE
    call ay_select_register
    call ay_write_register

    ; Громкость регистр( 9)
    ld a, 9
    jr @set_volume

@ch_a:
    ; Частота регистры( 0-1)
```

```
    ld a, e
    ld c, AY_REG_TONE_A_FINE
    call ay_select_register
    call ay_write_register
    ld a, d
    ld c, AY_REG_TONE_A_COARSE
    call ay_select_register
    call ay_write_register

    ; Громкость регистр( 8)
    ld a, 8

@set_volume:
    ; Устанавливаем громкость
    call ay_select_register
    pop bc           ; Восстанавливаем C громкость()
    push bc
    ld a, c
    and 0x0F         ; 4 бита громкости
    call ay_write_register

    xor a  ; Успех
    jr @exit

@invalid_channel:
    ld a, 3  ; AUDIO_ERR_INVALID_CHANNEL
    scf

@exit:
    pop hl
    pop de
    pop bc
    pop af
    ret

PUBLIC _audio_ay8910_stop
_audio_ay8910_stop:
    ; A = канал
    cp 255  ; AUDIO_CHANNEL_ALL
    jr z, @stop_all

    ; Остановить конкретный канал
    call _get_channel_bits

    ; Получить текущее значение регистра 7
    ld a, AY_REG_ENABLE
    call ay_select_register
    call ay_read_shadow

    ; Выключить тон установить( бит)
    or c
    call ay_write_register
```

```
    ; Установить громкость в 0
    call _get_volume_register
    call ay_select_register
    xor a
    call ay_write_register

    jr @exit

@stop_all:
    ; Выключить все тоны
    ld a, AY_REG_ENABLE
    call ay_select_register
    ld a, 0x38
    call ay_write_register

    ; Сбросить громкость на всех каналах
    ld b, 3
    ld c, AY_REG_VOLUME_A
@stop_loop:
    ld a, c
    call ay_select_register
    xor a
    call ay_write_register
    inc c
    djnz @stop_loop

@exit:
    xor a
    ret

; Остальные методы реализованы аналогично...
; Здесь[ будут реализации остальных методов по аналогии с play_note]

; Для краткости привожу только шаблоны остальных методов:

PUBLIC _audio_ay8910_set_volume
_audio_ay8910_set_volume:
    ; A = канал, B = громкость
    ret

PUBLIC _audio_ay8910_set_noise_type
_audio_ay8910_set_noise_type:
    ; A = тип шума
    ret

PUBLIC _audio_ay8910_set_noise_custom
_audio_ay8910_set_noise_custom:
    ; A = период шума
    ret

PUBLIC _audio_ay8910_enable_noise
_audio_ay8910_enable_noise:
    ; A = канал, B = enable
```

```
    ret

PUBLIC _audio_ay8910_set_envelope
_audio_ay8910_set_envelope:
    ; A = форма, B = скорость
    ret

PUBLIC _audio_ay8910_enable_envelope
_audio_ay8910_enable_envelope:
    ; A = канал, B = enable
    ret

PUBLIC _audio_ay8910_enable_tone
_audio_ay8910_enable_tone:
    ; A = канал, B = enable
    ret

PUBLIC _audio_ay8910_read_register
_audio_ay8910_read_register:
    ; A = регистр
    ret

PUBLIC _audio_ay8910_write_register
_audio_ay8910_write_register:
    ; A = значение, C = регистр
    ret

PUBLIC _audio_ay8910_play_chord
_audio_ay8910_play_chord:
    ; A = note1, B = note2, C = note3, D = volume
    ret

PUBLIC _audio_ay8910_beep
_audio_ay8910_beep:
    ; A = нота, HL = длительность
    ret
```

---

18. APPENDIX B: POLYMORPHIC DRIVER EXAMPLE

```
;; Example: Two video driver implementations

;; 1. Monochrome driver VTable
__video_mono_vtable:
    jp mono_detect
    jp mono_init
    jp mono_deinit
    jp mono_get_info
    jp mono_command
    jp mono_draw_pixel
    jp mono_draw_line
    ; ... additional methods
```

```
;; 2. Color driver VTable
__video_color_vtable:
    jp color_detect
    jp color_init
    jp color_deinit
    jp color_get_info
    jp color_command
    jp color_draw_pixel
    jp color_draw_line
    ; ... same method layout

;; 3. Switching between implementations
switch_video_mode:
    ; Check which mode to use
    ld a, (current_mode)
    cp MODE_COLOR
    jr z, .switch_to_color

    ; Switch to monochrome
    ld a, CMD_SWITCH_DRIVER
    ld hl, __video_mono_vtable
    call __video_command
    ret

.switch_to_color:
    ; Switch to color
    ld a, CMD_SWITCH_DRIVER
    ld hl, __video_color_vtable
    call __video_command
    ret
```

---

19. APPENDIX C: MEMORY CONTEXT EXAMPLE

```
;; Complete memory context usage example

;; 1. Define driver context
video_context:
    db 0xA5         ; Hash: slot ^ bank0 ^ bank1 ^ bank2 ^ bank3
    db 1            ; Slot register: slot 1
    db 0x10         ; Window 0: video code bank
    db 0x20         ; Window 1: video buffer bank
    db 0x30         ; Window 2: palette bank
    db 0x01         ; Window 3: system bank
    db 0            ; Flags
    db 0            ; Reserved

;; 2. Context-aware driver wrapper
video_draw_pixel_safe:
    ; Save current context, apply video context
    push hl
```

```
    ld hl, video_context
    call ensure_memory_context
    pop hl

    ; Call actual implementation
    call video_draw_pixel_impl

    ; Restore previous context
    call restore_previous_context
    ret

;; 3. System ensures context before VTable calls
__video_draw_pixel:
    jp video_draw_pixel_safe
```
