RTL Design Sherpa

APB HPET Micro-Architecture Specification 1.0

January 4, 2026

Table of Contents

List of Figures

No figures in this document.

List of Tables

No tables in this document.

List of Waveforms

No waveforms in this document.

1 Hpet Mas Index

Generated: 2026-01-04

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

1.0.1 APB HPET - Overview

1.0.1.1 Introduction

The APB High Precision Event Timer (HPET) is a configurable multi-timer peripheral designed for precise timing and event generation in embedded systems. It provides up to 8 independent hardware timers with one-shot and periodic modes, accessible via APB interface with optional clock domain crossing support.

APB HPET Block Diagram

APB HPET Block Diagram

1.0.1.2 Key Features

- **Multiple Independent Timers**: 2, 3, or 8 configurable hardware timers per instance
- **64-bit Main Counter**: High-resolution timestamp with configurable clock source
- **64-bit Comparators**: Long-duration timing support (up to 2^64-1 clock cycles)
- **Dual Operating Modes**:
  - **One-shot**: Timer fires once when counter reaches comparator value
  - **Periodic**: Timer auto-reloads and fires repeatedly at fixed intervals
- **Dynamic Mode Switching**: Switch between one-shot and periodic modes without reset
- **APB Interface**: Standard AMBA APB4 compliant register interface
- **Clock Domain Crossing**: Optional CDC support for independent APB and timer clocks
- **PeakRDL Integration**: Register map generated from SystemRDL specification

- **Per-Timer Write Data Buses**: Dedicated data paths prevent timer corruption
- **Individual Interrupts**: Separate interrupt output per timer with W1C status clearing

1.0.1.3 Applications

Real-Time Operating Systems: - System tick generation for RTOS schedulers - Watchdog timer implementation - Task deadline enforcement - Periodic interrupt generation

Performance Profiling: - High-resolution timestamp source - Code execution timing - Cache miss profiling - Inter-event timing measurement

Multi-Rate Timing: - Multiple simultaneous timing domains - Independent periodic tasks - Asynchronous event generation - Programmable pulse generation

Industrial Control: - PWM generation base timer - Motor control timing - Sensor sampling intervals - Control loop timing

1.0.1.4 Design Philosophy

Configurability: The HPET component prioritizes configurability to support diverse use cases. Timer count, vendor ID, and CDC enablement are all parameterizable at synthesis time, allowing customization for specific applications without RTL changes.

Reliability: Extensive testing (5/6 configurations at 100% pass rate) validates core functionality. The design includes per-timer data buses to prevent corruption and comprehensive error detection in configuration registers.

Standards Compliance: - **APB Protocol**: Full AMBA APB4 specification compliance - **PeakRDL**: Industry-standard SystemRDL for register generation - **Reset Convention**: Consistent active-low asynchronous reset (`presetn`)

Reusability: Clean module hierarchy and well-defined interfaces enable easy integration. Optional CDC support allows flexible clock domain configuration without design changes.

1.0.1.5 Comparison with IA-PC HPET

The APB HPET draws architectural inspiration from the IA-PC HPET specification (Intel/Microsoft) but is **not** a drop-in replacement. Key differences:

Feature	IA-PC HPET	APB HPET
Interface	Memory-mapped	AMBA APB4
Timer Count	Up to 256	2, 3, or 8 (configurable)
FSB Delivery	Supported	Not supported

Feature	IA-PC HPET	APB HPET
Legacy Replacement	PIT/RTC emulation	Not supported
Counter Size	64-bit mandatory	64-bit
Comparator Size	64-bit or 32-bit	64-bit only
Clock Source	10 MHz minimum	User-configurable
Vendor ID	Read from capability	Configurable parameter

Retained Concepts: - 64-bit free-running counter - One-shot and periodic timer modes - Write-1-to-clear interrupt status - Capability register for hardware discovery

Removed Features: - FSB interrupt delivery (use dedicated IRQ signals) - Legacy PIT/RTC replacement (not needed in modern designs) - Main counter period configuration (use clock divider instead)

1.0.1.6 Performance Characteristics

Timing Accuracy: - Counter increment: Every HPET clock cycle (deterministic) - Timer fire latency: 1 HPET clock cycle from counter match - Interrupt assertion: Combinational (same cycle as timer fire)

Register Access Latency: - No CDC: 2 APB clock cycles (APB protocol minimum) - With CDC: 4-6 APB clock cycles (handshake synchronization overhead)

Resource Utilization (Post-Synthesis Estimates): - 2-timer (no CDC): ~500 LUTs, ~300 flip-flops - 3-timer (no CDC): ~650 LUTs, ~400 flip-flops - 8-timer (with CDC): ~1200 LUTs, ~800 flip-flops

Scalability: The design scales linearly with timer count. Each additional timer adds approximately: - 150 LUTs (comparator, control logic, interrupt generation) - 100 flip-flops (timer state, configuration registers) - Minimal timing impact (no critical path through timer array)

1.0.1.7 Verification Status

Test Coverage: 5 of 6 configurations achieve 100% test pass rate

Configuration	Basic	Medium	Full	Overall
2-timer Intel-like (no CDC)	4/4 ✓	5/5 ✓	3/3 ✓	12/12 ✓
3-timer AMD-like (no CDC)	4/4 ✓	5/5 ✓	3/3 ✓	12/12 ✓

Configuration	Basic	Medium	Full	Overall
8-timer custom (no CDC)	4/4 ✓	5/5 ✓	2/3 ⚠️	11/12 ⚠️
2-timer Intel-like (CDC)	4/4 ✓	5/5 ✓	3/3 ✓	12/12 ✓
3-timer AMD-like (CDC)	4/4 ✓	5/5 ✓	3/3 ✓	12/12 ✓
8-timer custom (CDC)	4/4 ✓	5/5 ✓	3/3 ✓	12/12 ✓

Known Issue: 8-timer non-CDC “All Timers Stress” test has timeout issue (minor, likely test configuration)

Test Levels: - **Basic (4 tests)**: Register access, enable/disable, counter operation, interrupt generation - **Medium (5 tests)**: Periodic mode, multiple timers, 64-bit features, mode switching - **Full (3 tests)**: All timers stress, CDC validation, edge case coverage

See: `IMPLEMENTATION_STATUS.md` for complete test results

1.0.1.8 Development Status

Status: ✓ Production Ready

Completed Features: - ✓ One-shot timer mode - ✓ Periodic timer mode - ✓ Timer mode switching - ✓ 64-bit counter read/write - ✓ 64-bit comparators - ✓ Multiple independent timers - ✓ Clock domain crossing (optional) - ✓ PeakRDL register generation - ✓ Per-timer write data buses (corruption fix) - ✓ Comprehensive test suite (3-level hierarchy)

Outstanding Items: - ⚠️ 8-timer stress test timeout (minor, likely test configuration)

Future Enhancements (Not Planned): - Comparator readback (currently write-only) - FSB interrupt delivery (use dedicated IRQ signals) - Legacy mode emulation (not needed in modern designs) - 64-bit atomic counter reads (current implementation requires two 32-bit reads)

1.0.1.9 Documentation Organization

This specification document is organized as follows:

- **Chapter 1 (this chapter)**: Overview, features, applications

- **Chapter 2**: Detailed block specifications (hpet_core, config_regs, PeakRDL integration)
- **Chapter 3**: Interface specifications (APB, HPET clock, interrupts)
- **Chapter 4**: Programming model (initialization, configuration, use cases)
- **Chapter 5**: Register definitions (address map, field descriptions)

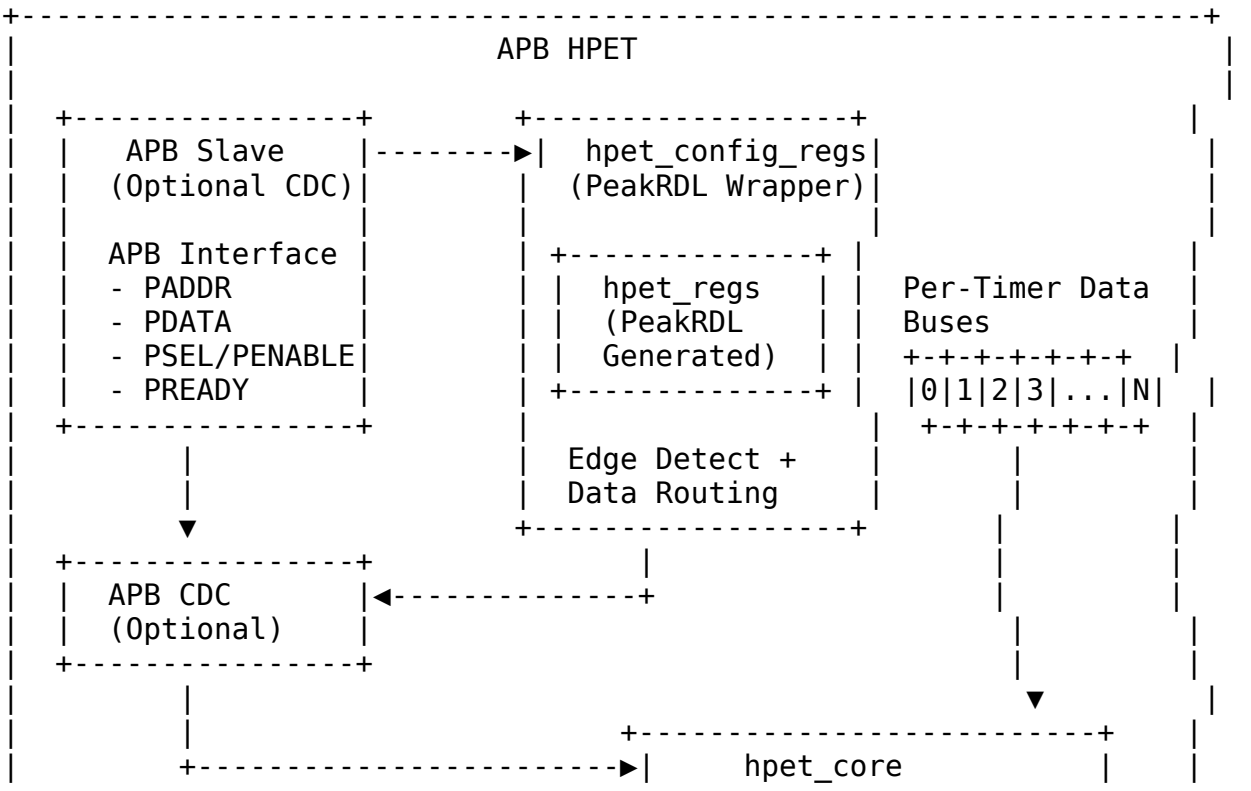
Related Documentation: - `../../PRD.md` - Product Requirements Document - `../../CLAUDE.md` - AI integration guide - `../../TASKS.md` - Development task tracking - `../IMPLEMENTATION_STATUS.md` - Test results and validation status

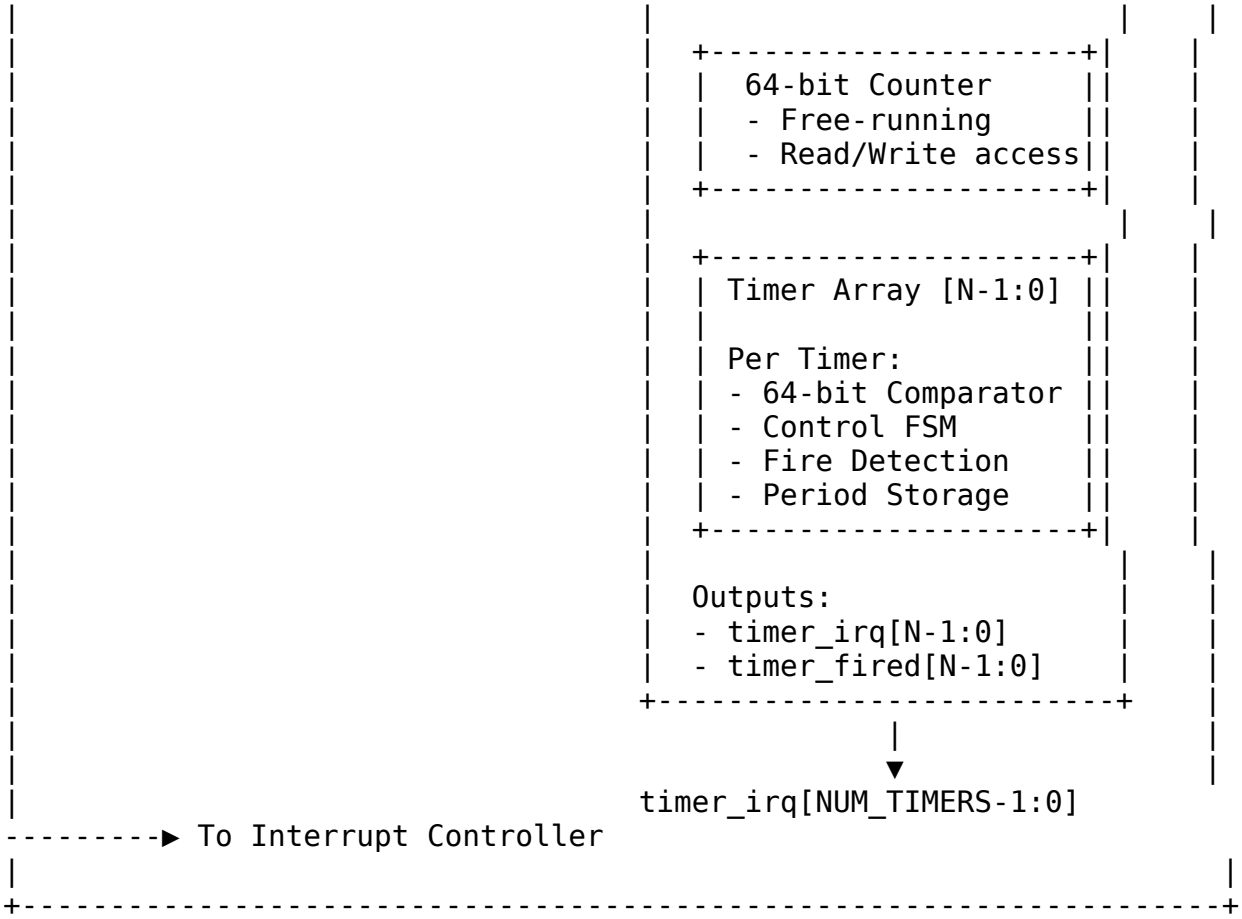
---

Next: Chapter 1.2 - Architecture

1.0.2 APB HPET - Architecture

1.0.2.1 High-Level Block Diagram





1.0.2.2 Module Hierarchy

```
apb_hpet (Top Level)
+-- apb_slave (OR apb_slave_cdc if CDC_ENABLE=1)
|   +-- APB protocol handling
|   +-- Read/write transaction management
|   +-- Optional clock domain crossing
|
+-- hpet_config_regs (Register Wrapper)
|   +-- hpet_regs (PeakRDL Generated)
|   |   +-- HPET_CONFIG register
|   |   +-- HPET_STATUS register (W1C)
|   |   +-- HPET_COUNTER_LO/HI registers
|   |   +-- HPET_CAPABILITIES register (RO)
|   |   +-- TIMER[i]_* registers (per-timer)
|   |
|   +-- edge_detect (x NUM_TIMERS) - Write strobe generation
|   +-- Per-timer data bus routing (corruption prevention)
|
+-- hpet_core (Timer Logic)
    +-- 64-bit main counter (r_main_counter)
```

```
    +-- Timer array [NUM_TIMERS-1:0]
    |   +-- 64-bit comparator (r_timer_comparator[i])
    |   +-- 64-bit period storage (r_timer_period[i])
    |   +-- Timer control FSM (one-shot vs periodic)
    |   +-- Fire detection logic
    +-- Counter increment logic
    +-- Comparator match detection
    +-- Interrupt generation
```

1.0.2.3 Data Flow

1.0.2.3.1 Write Transaction Flow (APB -> HPET Core)

```
1. APB Master Write
   |
   ▼
2. APB Slave (or APB CDC)
   - Protocol handling
   - Clock domain crossing (if enabled)
   |
   ▼
3. hpet_regs (PeakRDL)
   - Register decoding
   - Field updates
   - Software access flags (swacc, swmod)
   |
   ▼
4. hpet_config_regs
   - Edge detection on swacc signals
   - Generate write strobes (timer_comparator_wr[i])
   - Route per-timer data buses
   |
   ▼
5. hpet_core
   - Update counter (if HPET_COUNTER write)
   - Update comparator (if TIMER_COMPARATOR write)
   - Update control (if TIMER_CONFIG write)
   - Clear interrupt (if HPET_STATUS write with W1C)
```

1.0.2.3.2 Read Transaction Flow (HPET Core -> APB)

```
1. APB Master Read
   |
   ▼
2. APB Slave (or APB CDC)
   - Protocol handling
   - Read data synchronization (if CDC)
   |
   ▼
3. hpet_regs (PeakRDL)
```

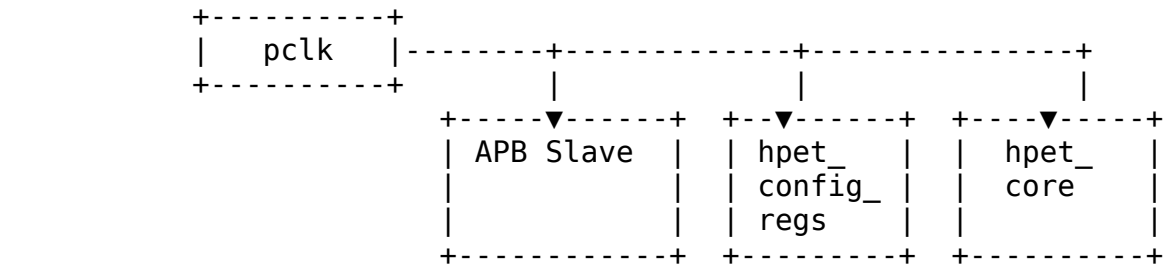
```
   - Address decode
   - Multiplex read data from hardware interface (hwif)
   |
   ▼
4. hpet_config_regs
   - Connect hpet_core signals to hwif read ports
   |
   ▼
5. hpet_core
   - Provide counter value
   - Provide timer configuration
   - Provide status flags
   |
   ▼
6. APB Slave returns PRDATA to master
```

1.0.2.3.3 Timer Operation Flow

```
1. Counter Increment (every hpet_clk)
   r_main_counter <= r_main_counter + 1
   |
   ▼
2. Comparator Match Detection (for each timer i)
   timer_match[i] = (r_main_counter >= r_timer_comparator[i])
   |
   ▼
3. Timer Fire Logic
   |
   +- One-Shot Mode:
   |  - Fire when match first detected
   |  - Stay idle until reconfigured
   |  - Assert timer_irq[i]
   |
   +- Periodic Mode:
      - Fire when match detected
      - Auto-increment comparator:
        r_timer_comparator[i] <= r_timer_comparator[i] +
r_timer_period[i]
      - Assert timer_irq[i]
      - Repeat
   |
   ▼
4. Interrupt Status Update
   HPET_STATUS[i] <= 1 (sticky until software clears via W1C)
   |
   ▼
5. Interrupt Output
   timer_irq[i] = HPET_STATUS[i] (combinational)
```

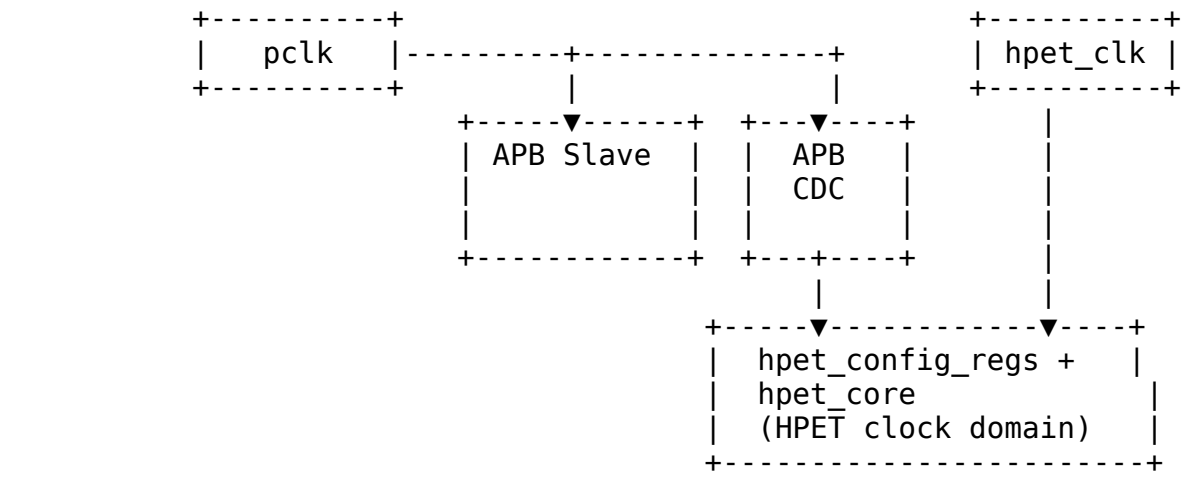
1.0.2.4 Clock Domains

Synchronous Mode (CDC_ENABLE = 0):



Note: pclk = hpet_clk (same clock domain)

Asynchronous Mode (CDC_ENABLE = 1):



Note: pclk and hpet_clk are asynchronous, CDC required

1.0.2.5 Reset Domains

Reset Signals: - `presetn` - APB reset (active-low, asynchronous) - `hpet_rst_n` - HPET reset (active-low, asynchronous)

Reset Behavior:

Signal	Reset Domain	Reset Value	Notes
`r_main_counter`	hpet_clk	64'h0	Counter reset to zero
`r_timer_comparator[i]`	hpet_clk	64'h0	Comparators reset to

Signal	Reset Domain	Reset Value	Notes
			zero
r_timer_period[i]	hpet_clk	64'h0	Period storage reset
HPET_CONFIG	pclk	Disabled	Global enable cleared
HPET_STATUS	pclk	8'h0	All interrupt flags cleared
TIMER[i]_CONFIG	pclk	Disabled	All timers disabled

Reset Sequence:

```
// APB domain reset
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn) begin
        // Reset APB-accessible registers
        HPET_CONFIG <= '0;
        HPET_STATUS <= '0;
        for (int i = 0; i < NUM_TIMERS; i++) begin
            TIMER_CONFIG[i] <= '0;
        end
    end
end

// HPET domain reset
always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n) begin
        // Reset timer logic
        r_main_counter <= 64'h0;
        for (int i = 0; i < NUM_TIMERS; i++) begin
            r_timer_comparator[i] <= 64'h0;
            r_timer_period[i] <= 64'h0;
            r_timer_fired[i] <= 1'b0;
        end
    end
end
```

CDC Reset Coordination: When CDC is enabled, both reset signals must be properly synchronized and coordinated to prevent metastability and ensure clean initialization.

1.0.2.6 Per-Timer Data Bus Architecture

Problem: Initial implementation had timer corruption due to shared data bus

Root Cause:

```
// ✗ WRONG: Shared data bus for all timers
wire [63:0] timer_comparator_data;  // Single 64-bit bus

// Multiple timers try to sample from same bus
always_ff @(posedge hpet_clk) begin
    if (timer_comparator_wr[0]) r_timer_comparator[0] <=
timer_comparator_data;
    if (timer_comparator_wr[1]) r_timer_comparator[1] <=
timer_comparator_data;
    if (timer_comparator_wr[2]) r_timer_comparator[2] <=
timer_comparator_data;
    // If write strobes overlap, wrong timer gets wrong data!
end
```

Solution: Per-timer dedicated data buses

```
// ✓ CORRECT: Dedicated data bus per timer
wire [63:0] timer_comparator_data [NUM_TIMERS-1:0];  // Array of 64-
bit buses

// Each timer has dedicated data path
always_ff @(posedge hpet_clk) begin
    if (timer_comparator_wr[0]) r_timer_comparator[0] <=
timer_comparator_data[0];
    if (timer_comparator_wr[1]) r_timer_comparator[1] <=
timer_comparator_data[1];
    if (timer_comparator_wr[2]) r_timer_comparator[2] <=
timer_comparator_data[2];
    // Each timer reads from its own dedicated bus - no corruption
possible
end
```

Implementation in hpet_config_regs.sv:

```
// Dedicated data buses prevent corruption
assign timer_comparator_data[0] = {hwif.timer0_comparator_hi.value,
                                   hwif.timer0_comparator_lo.value};
assign timer_comparator_data[1] = {hwif.timer1_comparator_hi.value,
                                   hwif.timer1_comparator_lo.value};
assign timer_comparator_data[2] = {hwif.timer2_comparator_hi.value,
```

```
                                        hwif.timer2_comparator_lo.value};
// ... one data bus per timer
```

Verification: All timer corruption issues resolved after per-timer bus implementation

1.0.2.7 Parameterization

Compile-Time Parameters:

Parameter	Type	Default	Range	Description
NUM_TIMERS	int	2	2, 3, 8	Number of independent timers
VENDOR_ID	int (16-bit)	0x8086	0x0000-0xFFFF	Vendor identification
REVISION_ID	int (16-bit)	0x0001	0x0000-0xFFFF	Hardware revision
CDC_ENABLE	bit	0	0, 1	Enable clock domain crossing
ADDR_WIDTH	int	12	>= 12	APB address bus width
DATA_WIDTH	int	32	32	APB data bus width (fixed)

Derived Parameters:

```
localparam int TIMER_ADDR_OFFSET = 32'h20;  // 32-byte stride per
timer
localparam int TIMER_REGS_START = 32'h100;  // Timer register base
address
```

Configuration Examples:

2-Timer “Intel-like” Configuration:

```
apb_hpet #(
    .NUM_TIMERS(2),
    .VENDOR_ID(16'h8086),   // Intel
    .REVISION_ID(16'h0001),
    .CDC_ENABLE(0)          // Synchronous clocks
) u_hpet_intel (...);
```

3-Timer “AMD-like” Configuration:

```
apb_hpet #(
    .NUM_TIMERS(3),
```

```
    .VENDOR_ID(16'h1022),   // AMD
    .REVISION_ID(16'h0002),
    .CDC_ENABLE(0)
) u_hpet_amd (...);
```

8-Timer Custom with CDC:

```
apb_hpet #(
    .NUM_TIMERS(8),
    .VENDOR_ID(16'hABCD),   // Custom vendor
    .REVISION_ID(16'h0010),
    .CDC_ENABLE(1)          // Asynchronous clocks
) u_hpet_custom (...);
```

1.0.2.8 Interface Summary

APB Interface: Standard AMBA APB4 - Address width: Configurable (default 12-bit for 4KB space) - Data width: Fixed 32-bit - Protocol: APB4 (with PREADY support)

HPET Clock Interface: Separate timer clock domain - Independent from APB clock (if CDC enabled) - Free-running 64-bit counter - Configurable clock frequency

Interrupt Interface: Per-timer dedicated outputs - `timer_irq[NUM_TIMERS-1:0]` - Active-high interrupt signals - Combinational output (driven by STATUS register) - W1C clearing via HPET_STATUS register

See: Chapter 3 - Interface Specifications for detailed signal descriptions

---

Next: Chapter 1.3 - Clocks and Reset

1.0.3 APB HPET - Clocks and Reset

1.0.3.1 Clock Domains

The APB HPET operates in one or two clock domains depending on CDC configuration:

1.0.3.1.1 Single Clock Domain (CDC_ENABLE = 0)

Configuration: - `pclk = hpet_clk` (same physical clock) - No clock domain crossing required - Lower latency (2 APB clock cycles for register access) - Simpler timing analysis

Use Cases: - System where APB and timer clocks are guaranteed synchronous - Resource-constrained designs (CDC overhead not needed) - Minimal latency requirements

1.0.3.1.2 Dual Clock Domains (CDC_ENABLE = 1)

Configuration: - `pclk` and `hpet_clk` are independent, asynchronous clocks - CDC synchronization required - Higher latency (4-6 APB clock cycles for register access) - More complex timing analysis

Use Cases: - System where APB runs at different frequency than timer clock - HPET clock derived from external crystal/oscillator - Power management scenarios (clock gating one domain)

1.0.3.2 Clock Specifications

1.0.3.2.1 APB Clock (`pclk`)

Purpose: APB interface protocol clock

Constraints: - Frequency: Typically 10-200 MHz (application-dependent) - Duty cycle: 50% ±10% - Jitter: < 5% of period - No specific minimum/maximum frequency enforced in RTL

Driven Blocks: - APB slave (or APB CDC wrapper) - PeakRDL register file - Register configuration logic

1.0.3.2.2 HPET Clock (`hpet_clk`)

Purpose: Timer counter increment and comparator evaluation

Constraints: - Frequency: User-configurable (typically 1-100 MHz) - Duty cycle: 50% ±10% - Jitter: < 2% of period (affects timer accuracy) - Must be stable and continuous when HPET enabled

Driven Blocks: - Main counter increment - Comparator match detection - Timer control FSMs - Interrupt generation logic

Timer Accuracy: Directly proportional to `hpet_clk` frequency and stability - 10 MHz -> 100ns resolution - 1 MHz -> 1µs resolution - 1 kHz -> 1ms resolution

1.0.3.3 Reset Domains

1.0.3.3.1 APB Reset (`presetn`)

Type: Asynchronous active-low reset

Scope: APB interface and configuration registers

Reset Behavior:

```
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn) begin
        // Global configuration
        HPET_CONFIG <= 32'h0;          // HPET disabled
```

```
        HPET_STATUS <= 32'h0;          // All interrupts cleared

        // Per-timer configuration
        for (int i = 0; i < NUM_TIMERS; i++) begin
            TIMER_CONFIG[i] <= 32'h0;  // Timer disabled
        end
    end
end
```

Reset Values: | Register | Reset Value | Description | |————-|————-|————-| | `HPET_CONFIG` | 32’h0 | Global disable, no legacy mapping | | `HPET_STATUS` | 32’h0 | All interrupt flags cleared | | `HPET_COUNTER_LO` | N/A | Write-only from APB domain | | `HPET_COUNTER_HI` | N/A | Write-only from APB domain | | `HPET_CAPABILITIES` | Read-only | Contains NUM_TIMERS, VENDOR_ID, REVISION_ID | | `TIMER[i]_CONFIG` | 32’h0 | Timer disabled, one-shot mode | | `TIMER[i]_COMPARATOR_LO` | N/A | Write-only | | `TIMER[i]_COMPARATOR_HI` | N/A | Write-only |

1.0.3.3.2 HPET Reset (hpet_rst_n)

Type: Asynchronous active-low reset

Scope: Timer counter and timer logic

Reset Behavior:

```
always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n) begin
        // Main counter
        r_main_counter <= 64'h0;

        // Per-timer state
        for (int i = 0; i < NUM_TIMERS; i++) begin
            r_timer_comparator[i] <= 64'h0;
            r_timer_period[i] <= 64'h0;
            r_timer_fired[i] <= 1'b0;
        end
    end
end
```

Reset Values: | Signal | Reset Value | Description | |———|————-|————-| | `r_main_counter` | 64’h0 | Counter starts at zero | | `r_timer_comparator[i]` | 64’h0 | Comparators cleared | | `r_timer_period[i]` | 64’h0 | Period storage cleared | | `r_timer_fired[i]` | 1’b0 | Fire flags cleared |

1.0.3.4 Reset Coordination

1.0.3.4.1 Synchronous Mode (CDC_ENABLE = 0)

Requirement: `presetn` and `hpet_rst_n` should be asserted/deasserted together

Recommended Connection:

```
assign hpet_rst_n = presetn;  // Same reset for both domains
```

Reset Sequence:

```
1. Assert presetn = 0 (also asserts hpet_rst_n = 0)
2. Hold for >= 10 clock cycles
3. Deassert presetn = 1 (also deasserts hpet_rst_n = 1)
4. Wait >= 5 clock cycles before first register access
```

1.0.3.4.2 Asynchronous Mode (CDC_ENABLE = 1)

Requirement: Both resets can be independent but must overlap during power-on

Recommended Sequence:

```
1. Assert both presetn = 0 and hpet_rst_n = 0
2. Hold presetn for >= 10 pclk cycles
3. Hold hpet_rst_n for >= 10 hpet_clk cycles
4. Deassert resets (order not critical, but both must be stable)
5. Wait for CDC handshake to stabilize (>= 6 pclk cycles)
6. Begin register accesses
```

Reset Timing Diagram (CDC Mode):

```
           +-----------------------------------
presetn    +                                  (>=10 pclk cycles in
reset)

                  +---------------------------------
hpet_rst_n        +                                  (>=10 hpet_clk cycles
in reset)

                            +-------------------------
APB Access                  + Safe to access       (Wait for CDC
stabilization)
```

1.0.3.5 Clock Domain Crossing Details

1.0.3.5.1 CDC Synchronization

When `CDC_ENABLE = 1`, the `apb_slave_cdc` module handles all clock domain crossing:

Write Path (pclk -> hpet_clk):

```
1. APB write on pclk
2. Command written to APB-side holding registers
3. Handshake synchronizer transfers command to hpet_clk domain
4. hpet_clk-side logic applies write to timer registers
```

```
5. Acknowledgment synchronized back to pclk
6. APB PREADY asserted (transaction complete)

Latency: 4-6 pclk cycles
```

Read Path (hpet_clk -> pclk):

```
1. APB read on pclk
2. Read request synchronized to hpet_clk
3. hpet_clk-side logic captures register data
4. Data synchronized back to pclk domain
5. APB PRDATA driven
6. APB PREADY asserted (transaction complete)

Latency: 4-6 pclk cycles
```

Metastability Protection: - All CDC signals pass through 2-stage synchronizers - Handshake protocol ensures data stability before sampling - No combinational paths cross clock domains

1.0.3.5.2 Counter Read Atomicity

Problem: 64-bit counter spans two 32-bit APB registers

Non-Atomic Read Sequence:

```
1. Read HPET_COUNTER_LO -> captures lower 32 bits
2. Counter increments (may overflow from 0xFFFFFFFF to 0x00000000)
3. Read HPET_COUNTER_HI -> captures upper 32 bits (now incremented!)
4. Result: Lower 32 bits from time T, upper 32 bits from time T+1
```

Software Workaround (Overflow Detection):

```c
uint64_t read_hpet_counter(void) {
    uint32_t hi1, hi2, lo;

    do {
        hi1 = read_reg(HPET_COUNTER_HI);
        lo  = read_reg(HPET_COUNTER_LO);
        hi2 = read_reg(HPET_COUNTER_HI);
    } while (hi1 != hi2);  // Retry if overflow detected

    return ((uint64_t)hi2 << 32) | lo;
}
```

Note: Hardware atomic read not implemented (future enhancement)

1.0.3.6 Clock Gating Considerations

APB Clock Gating: - Safe to gate `pclk` when no APB transactions pending - Must ensure APB master deasserts PSEL before gating - Gating has no effect on HPET timer operation (hpet_clk independent)

HPET Clock Gating: - **DO NOT gate `hpet_clk` while HPET enabled** (HPET_CONFIG[0] = 1) - Counter will stop incrementing -> timers will not fire - Safe to gate only when HPET_CONFIG[0] = 0 (disabled state)

Power Saving Strategy:

```
1. Disable HPET: Write HPET_CONFIG[0] = 0
2. Wait for any pending timer operations to complete
3. Gate hpet_clk
4. APB registers remain accessible (pclk still running)
5. To resume: Ungate hpet_clk, then write HPET_CONFIG[0] = 1
```

1.0.3.7 Timing Constraints

1.0.3.7.1 Setup/Hold Requirements

APB Interface (Synchronous):

```
Setup time:  2ns typical (technology-dependent)
Hold time:   1ns typical (technology-dependent)
```

HPET Clock (Asynchronous with CDC):

```
No setup/hold requirements between pclk and hpet_clk
CDC synchronizers handle all timing
```

1.0.3.7.2 Maximum Operating Frequencies

Technology-Dependent Estimates (Post-Synthesis): - APB clock: 200+ MHz (typical modern process) - HPET clock: 100+ MHz (limited by counter/comparator logic) - Clock domain crossing: Synchronizers support arbitrary frequency ratios

Recommended Operating Points: - APB clock: 10-100 MHz (typical SoC bus speeds) - HPET clock: 1-50 MHz (sufficient for most timing applications)

Next: Chapter 1.4 - Acronyms and Terminology

1.0.4 APB HPET - Acronyms and Terminology

1.0.4.1 Acronyms

Acronym	Full Term	Description
AMBA	Advanced Microcontroller Bus Architecture	ARM's on-chip interconnect specification
APB	Advanced Peripheral Bus	AMBA low-complexity peripheral bus protocol
CDC	Clock Domain Crossing	Synchronization between asynchronous clock domains
FSB	Front Side Bus	Legacy PC architecture bus (not supported in APB HPET)
FSM	Finite State Machine	Sequential logic controller
HPET	High Precision Event Timer	Multi-timer peripheral for precise timing
IA-PC	Intel Architecture - Personal Computer	PC platform specification (architectural reference)
IRQ	Interrupt Request	Hardware interrupt signal
PIT	Programmable Interval Timer	Legacy PC timer (8254-compatible)
RO	Read-Only	Register field cannot be written by software
RTC	Real-Time Clock	Calendar/time-of-day clock (not emulated by HPET)
RW	Read-Write	Register field can be read and written
SystemRDL	System Register Description Language	Industry-standard register specification language
W1C	Write-1-to-Clear	Register field cleared by writing 1, writing 0 has no effect
WO	Write-Only	Register field can only be

Acronym	Full Term	Description
		written, reads return undefined

1.0.4.2 Terminology

64-bit Counter: The main free-running counter that increments on every HPET clock cycle. Provides high-resolution timestamp and comparison base for all timers.

Comparator: Per-timer 64-bit value that defines when a timer should fire. Timer fires when main counter value becomes greater than or equal to comparator value.

Fire / Fired: Event when a timer's comparator matches the main counter value. In one-shot mode, timer fires once. In periodic mode, timer fires repeatedly.

One-Shot Mode: Timer operating mode where the timer fires once when the counter reaches the comparator value, then remains idle until reconfigured.

Periodic Mode: Timer operating mode where the timer fires repeatedly at fixed intervals. After each fire event, the comparator is automatically incremented by the period value.

Period: In periodic mode, the interval (in HPET clock cycles) between timer fires. Stored internally and used for auto-incrementing the comparator.

PeakRDL: Industry-standard toolchain for generating register files from SystemRDL specifications. Used to generate `hpet_regs.sv` from `hpet_regs.rdl`.

Per-Timer Data Bus: Dedicated 64-bit data path for each timer to prevent corruption when multiple timer registers are written in rapid succession.

Timer Corruption: Historical bug where shared data bus allowed one timer's configuration to overwrite another timer's configuration. Fixed by implementing per-timer dedicated data buses.

Write Strobe: Edge-detected pulse generated when software writes to a timer configuration register. Used to sample comparator and configuration data atomically.

1.0.4.3 Register Field Access Types

RO (Read-Only): - Software can read the field - Software writes are ignored - Hardware controls the value - Example: `HPET_CAPABILITIES` register

RW (Read-Write): - Software can read and write the field - Hardware may also update the value - Example: `HPET_CONFIG[0]` (enable bit)

WO (Write-Only): - Software can write the field - Software reads return undefined value - Hardware uses written value internally - Example: `HPET_COUNTER_LO/HI` (write from APB domain, read by HPET core)

W1C (Write-1-to-Clear): - Software writes 1 to clear the bit - Software writes 0 have no effect - Hardware can set the bit - Example: `HPET_STATUS` interrupt flags

1.0.4.4 Signal Naming Conventions

APB Signals: All APB signals use standard AMBA naming with `p` prefix: - `pclk` - APB clock - `presetn` - APB reset (active-low) - `paddr` - APB address bus - `psel` - APB select - `penable` - APB enable - `pwrite` - APB write enable - `pwdata` - APB write data - `pready` - APB ready - `prdata` - APB read data - `pslverr` - APB slave error

HPET Domain Signals: Timer-related signals use descriptive names: - `hpet_clk` - HPET timer clock - `hpet_rst_n` - HPET reset (active-low) - `timer_irq[N]` - Timer interrupt outputs - `r_main_counter` - Internal 64-bit counter - `r_timer_comparator[i]` - Per-timer comparator value - `r_timer_period[i]` - Per-timer period value

Prefix Conventions: - `r_` - Registered (flip-flop) signal - `w_` - Wire (combinational) signal - `cfg_` - Configuration input - `hwif_` - PeakRDL hardware interface signal

1.0.4.5 Common Abbreviations in Code

Abbreviation	Meaning	Example
`cfg`	Configuration	`cfg_initial_credit`
`cmp`	Comparator	`timer_cmp_data`
`wr`	Write	`timer_comparator_wr`
`rd`	Read	`counter_rd_data`
`hi`	High (upper 32 bits)	`HPET_COUNTER_HI`
`lo`	Low (lower 32 bits)	`HPET_COUNTER_LO`
`en`	Enable	`timer_en`
`irq`	Interrupt Request	`timer_irq`
`clr`	Clear	`status_clr`

Next: Chapter 1.5 - References

1.0.5 APB HPET - References

1.0.5.1 External Standards and Specifications

AMBA Protocol Specifications: - **AMBA APB Protocol Specification v2.0** - Publisher: ARM Limited - Document ID: IHI 0024C - URL: https://developer.arm.com/documentation/ihi0024/latest - Relevance: APB interface protocol specification

SystemRDL: - **SystemRDL 2.0 Specification** - Publisher: Accellera Systems Initiative - URL: https://www.accellera.org/downloads/standards/systemrdl - Relevance: Register description language for `hpet_regs.rdl`

- **PeakRDL Documentation**
  - Project: PeakRDL Register Description Language Compiler
  - URL: https://peakrdl.readthedocs.io/
  - Relevance: SystemRDL to SystemVerilog compiler tool

Architectural Reference (Not Specification Compliant): - **IA-PC HPET Specification 1.0a** - Publisher: Intel Corporation and Microsoft Corporation - Date: October 2004 - URL: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf - Relevance: Architectural inspiration (APB HPET is NOT IA-PC HPET compliant) - **Note:** Used as reference for timer concepts only. APB HPET uses APB interface (not memory-mapped), different register layout, and does not support legacy modes or FSB delivery.

1.0.5.2 Internal Project Documentation

Component-Specific Documentation: - PRD.md - Product Requirements Document - Complete functional requirements - Parameter specifications - Verification status

- CLAUDE.md - AI Integration Guide
  - Component architecture overview
  - Known issues and workarounds
  - Test methodology
- TASKS.md - Development Task Tracking
  - Active work items
  - Completed milestones
  - Future enhancements
- IMPLEMENTATION_STATUS.md - Test Results
  - Detailed test results per configuration
  - Pass/fail statistics
  - Root cause analysis

RTL Source Files: - `rtl/apb_hpet.sv` - Top-level wrapper module - `rtl/hpet_core.sv` - Core timer logic - `rtl/hpet_config_regs.sv` - Register wrapper - `rtl/hpet_regs.sv` - PeakRDL generated register file (from hpet_regs.rdl) - `rtl/hpet_regs_pkg.sv` - PeakRDL generated package

SystemRDL Specification: - `rtl/peakrdl/hpet_regs.rdl` - Register description - `rtl/peakrdl/README.md` - PeakRDL generation instructions

Testbench Files: - `dv/tbclasses/hpet_tb.py` - Main testbench class - `dv/tbclasses/hpet_tests_basic.py` - Basic test suite - `dv/tbclasses/hpet_tests_medium.py` - Medium test suite - `dv/tbclasses/hpet_tests_full.py` - Full test suite - `dv/tests/test_apb_hpet.py` - Test runner with pytest integration

Known Issues Documentation: - `known_issues/README.md` - Issue tracking overview - `known_issues/resolved/timer_cleanup_issue.md` - Timer corruption fix details

1.0.5.3 Repository-Wide Documentation

Root Documentation: - `/README.md` - Repository overview and setup - `/PRD.md` - Master project requirements - `/CLAUDE.md` - Repository-wide AI guidance

Framework Documentation: - `bin/CocoTBFramework/README.md` - Testbench framework overview - `bin/CocoTBFramework/CLAUDE.md` - Framework usage guide - `bin/CocoTBFramework/components/apb/README.md` - APB BFM documentation

Verification Architecture: - `docs/VERIFICATION_ARCHITECTURE_GUIDE.md` - Complete verification patterns - Three-layer architecture (TB + Scoreboard + Test) - Queue-based vs memory model verification - Mandatory testbench methods

1.0.5.4 Related RTL Components

APB Infrastructure: - `rtl/amba/apb/apb_slave.sv` - Standard APB slave - `rtl/amba/apb/apb_slave_cdc.sv` - APB slave with clock domain crossing - `rtl/amba/adapters/peakrdl_to_cmdrsp.sv` - PeakRDL adapter

Clock Domain Crossing: - `rtl/amba/shared/cdc_handshake.sv` - CDC handshake synchronizer - `rtl/common/sync_2ff.sv` - 2-stage synchronizer - `rtl/common/sync_pulse.sv` - Pulse synchronizer

Common Utilities: - `rtl/common/edge_detect.sv` - Edge detection logic (used for write strobes) - `rtl/common/counter_bin.sv` - Binary counter (similar to HPET main counter)

1.0.5.5 Design Tools

Simulation: - Verilator 5.0+ - RTL simulator - CocoTB 1.9+ - Python testbench framework - pytest 7.0+ - Test runner and parametrization

Register Generation: - PeakRDL-regblock 0.17+ - SystemRDL to SystemVerilog compiler - PeakRDL 1.0+ - SystemRDL front-end

Waveform Viewing: - GTKWave - VCD waveform viewer - GTKW files available in `dv/GTKW/` directory

1.0.5.6 Industry Best Practices References

RTL Coding: - *Synthesis and Simulation Design Guide* - Xilinx UG901 - Best practices for RTL coding style - Clock domain crossing guidelines - Reset strategies

- *RTL Modeling with SystemVerilog for Simulation and Synthesis* - Stuart Sutherland
  - SystemVerilog coding guidelines
  - Finite state machine design patterns

Verification: - *Writing Testbenches using SystemVerilog* - Janick Bergeron - Testbench architecture patterns - Functional coverage methodology

- *Verification Methodology Manual for SystemVerilog* - Janick Bergeron et al.
  - UVM-like verification patterns
  - Coverage-driven verification

AMBA Protocols: - *AMBA Design Kit (ADK)* - ARM - Reference implementations - Protocol checkers - Example testbenches

1.0.5.7 Version Control and Issue Tracking

Git Repository: - Main branch: Production-ready code - Feature branches: Active development - Commit history: Detailed change log

Issue Labels: - `bug` - Functional defects - `enhancement` - New features - `documentation` - Documentation updates - `testing` - Test infrastructure improvements

1.0.5.8 Related Projects

RTL Design Sherpa Components: - APB HPET (this component) - AMBA AXI4 Monitors (`rtl/amba/`) - RAPIDS DMA Engine (`projects/components/rapids/`) - Delta Network Arbiter (`projects/components/delta/`)

External Dependencies: - None - APB HPET is fully self-contained within RTL Design Sherpa

---

Next: Chapter 2 - Blocks

1.0.6 APB HPET Blocks - Overview

1.0.6.1 Block Hierarchy

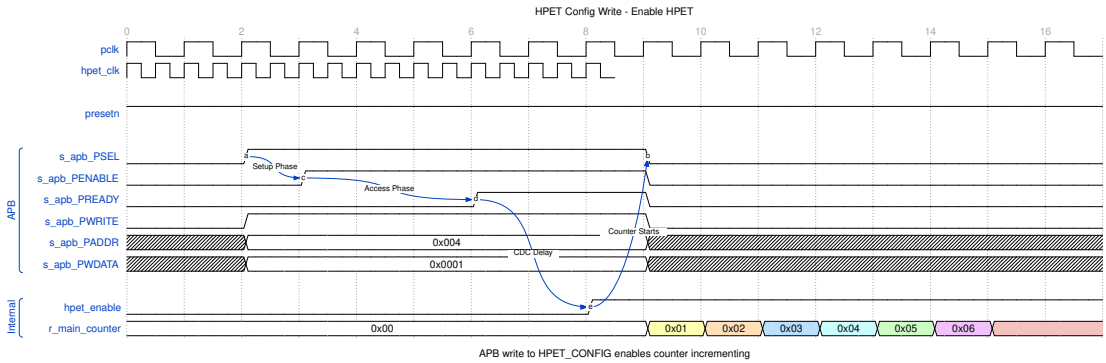
The APB HPET component consists of four primary SystemVerilog modules organized in a hierarchical structure:

```
apb_hpet (Top Level)
+-- APB Slave Interface
|   +-- apb_slave.sv (CDC_ENABLE=0) OR
|   +-- apb_slave_cdc.sv (CDC_ENABLE=1)
|
+-- hpet_config_regs (Register Wrapper)
|   +-- hpet_regs (PeakRDL Generated)
|   |   +-- Register File Logic
|   |
|   +-- Mapping Logic
|       +-- Per-Timer Data Buses
|       +-- Edge Detection
|       +-- Counter Write Capture
|
+-- hpet_core (Timer Logic)
    +-- 64-bit Free-Running Counter
    +-- Per-Timer Comparators [NUM_TIMERS]
    +-- Fire Detection Logic [NUM_TIMERS]
    +-- Interrupt Generation [NUM_TIMERS]
```

1.0.6.2 Timer Operation Waveforms

1.0.6.2.1 Configuration Write

When software writes to HPET_CONFIG to enable the timer, the enable signal propagates through the register file to the core.

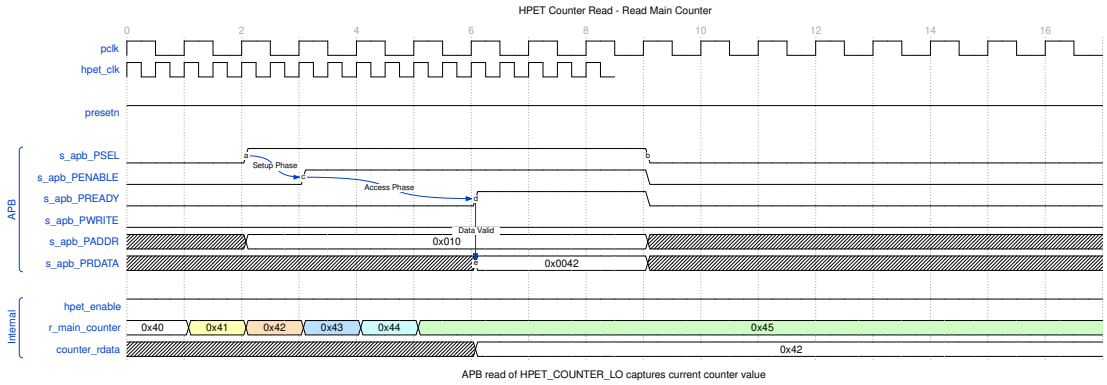


HPET Config Write

The APB write to address 0x004 (HPET_CONFIG) sets `hpet_enable`, which starts the main counter incrementing.

1.0.6.2.2 Counter Read

Reading the main counter returns the current 64-bit counter value.

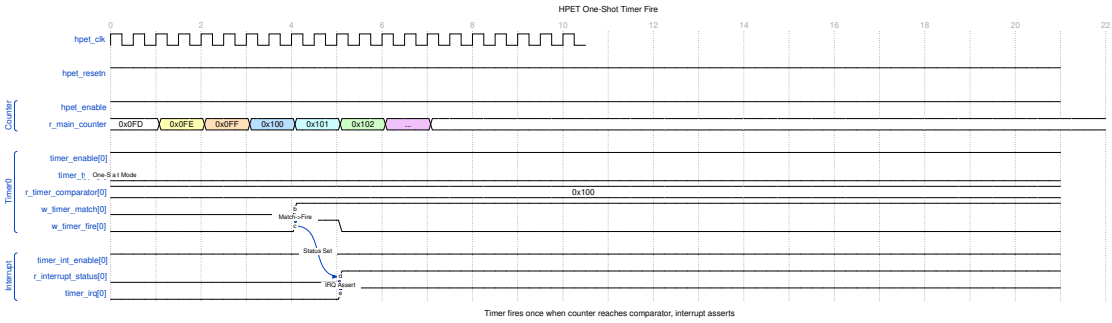


HPET Counter Read

The counter value is captured during the APB read transaction and returned on PRDATA.

1.0.6.2.3 One-Shot Timer Fire

In one-shot mode, the timer fires once when the counter reaches the comparator value.

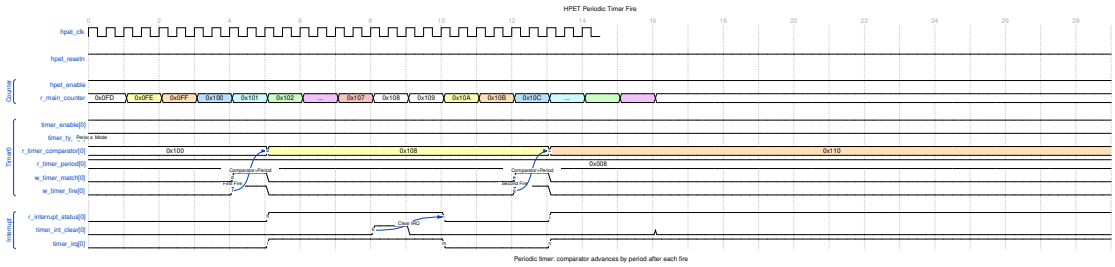


HPET One-Shot Timer Fire

When `r_main_counter` equals `r_timer_comparator[0]`, the match signal asserts, triggering `w_timer_fire[0]`. The interrupt output `timer_irq[0]` asserts and remains active until software clears it.

1.0.6.2.4 Periodic Timer Fire

In periodic mode, the timer fires repeatedly, automatically adding the period to the comparator.

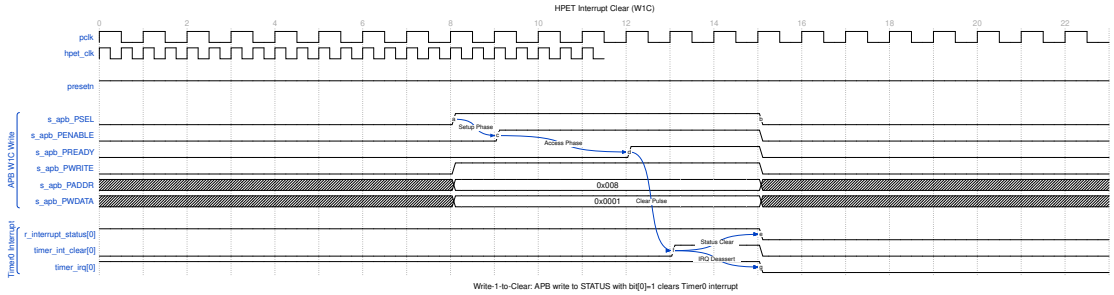


HPET Periodic Timer Fire

After each fire event, the comparator is updated: `comparator += period`. This allows continuous periodic interrupts without software intervention.

1.0.6.2.5 Interrupt Clear (W1C)

Software clears timer interrupts by writing 1 to the corresponding bit in HPET_STATUS.

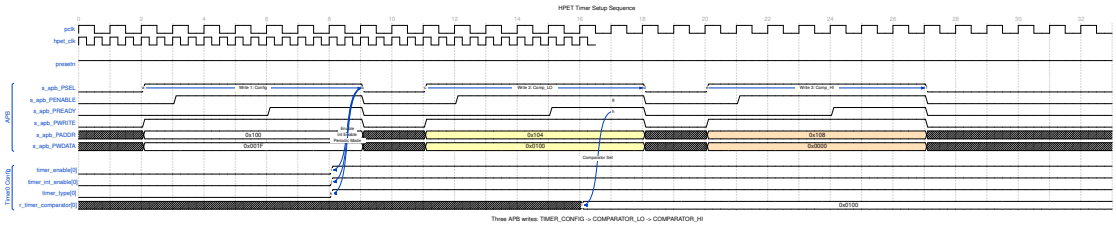


HPET Interrupt Clear

The W1C (Write-1-to-Clear) mechanism allows atomic clearing of individual timer interrupts.

1.0.6.2.6 Timer Setup Sequence

Configuring a timer requires multiple APB writes: config register, then comparator low/high words.

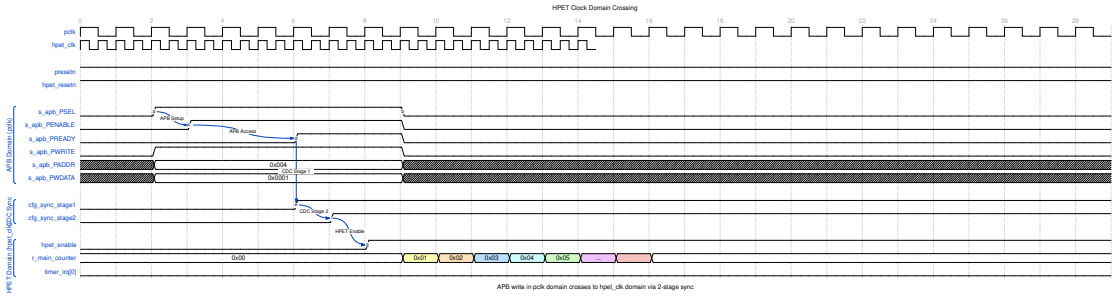


HPET Timer Setup

The sequence shows three consecutive writes: 1. TIMER_CONFIG (0x100): Enable, interrupt enable, periodic mode 2. TIMER_COMPARATOR_LO (0x104): Lower 32 bits of comparator 3. TIMER_COMPARATOR_HI (0x108): Upper 32 bits of comparator

1.0.6.2.7 Clock Domain Crossing (CDC Mode)

When CDC_ENABLE=1, APB transactions cross from pclk to hpet_clk domain via 2-stage synchronizers.



HPET CDC Crossing

The diagram shows the latency introduced by CDC synchronization. Configuration changes in the APB domain take 2-3 hpet_clk cycles to affect the timer core

---

1.0.6.3 Module Responsibilities

1.0.6.3.1 1. apb_hpet (Top Level Integration)

File: `rtl/apb_hpet.sv` **Purpose:** System integration and CDC selection

Responsibilities: - Instantiates APB slave with or without CDC based on `CDC_ENABLE` parameter - Routes signals between APB interface and configuration registers - Exposes timer interrupts to system - Provides unified external interface

Key Features: - Conditional CDC instantiation (generate block) - Clock domain management - Parameter propagation to child modules - Single-point configuration

1.0.6.3.2 2. hpet_config_regs (Register Wrapper)

File: `rtl/hpet_config_regs.sv` **Purpose:** Bridge between PeakRDL registers and HPET core

Responsibilities: - Instantiates PeakRDL-generated register file - Maps PeakRDL hardware interface to HPET core signals - Implements per-timer dedicated data buses (corruption fix) - Detects register write edges for control strobes - Handles 32-bit to 64-bit register combining

Key Features: - Per-timer data buses prevent configuration corruption - Edge detection for write strobes (not level) - Counter write capture from APB domain - W1C interrupt clearing support

1.0.6.3.3 3. hpet_regs (PeakRDL Generated)

File: `rtl/hpet_regs.sv` **Purpose:** Auto-generated register file from SystemRDL specification

Responsibilities: - Implements all HPET registers from RDL specification - Provides CPU interface (passthrough protocol) - Generates hardware interface structs - Handles field access types (RO, RW, W1C)

Key Features: - Single source of truth (hpet_regs.rdl) - Regeneratable from specification - Comprehensive field control - Standard passthrough CPU interface

1.0.6.3.4 4. hpet_core (Timer Logic)

File: `rtl/hpet_core.sv` **Purpose:** Core timer functionality and comparison logic

Responsibilities: - Implements 64-bit free-running counter - Manages per-timer comparators and periods - Detects counter match conditions - Generates timer fire events and interrupts - Handles one-shot vs periodic mode differences

Key Features: - Fully synchronous timer logic - Per-timer FSM (conceptual) - Automatic period reload (periodic mode) - Edge-based fire detection - Configurable timer count (2, 3, or 8 timers)

1.0.6.4 Data Flow Overview

1.0.6.4.1 APB Write Transaction Flow

```
APB Master
    ↓ PSEL, PENABLE, PADDR, PWDATA
APB Slave (or APB Slave CDC)
    ↓ cmd_valid, cmd_pwrite, cmd_paddr, cmd_pwdata
peakrdl_to_cmdrsp Adapter
    ↓ regblk_req, regblk_req_is_wr, regblk_addr, regblk_wr_data
hpet_regs (PeakRDL)
    ↓ hwif_out (register values)
hpet_config_regs (Mapping)
    ↓ timer_enable, timer_comparator_wr, timer_comparator_data[i]
hpet_core (Timer Logic)
    -> Counter/Comparator update
```

1.0.6.4.2 APB Read Transaction Flow

```
APB Master
    ↓ PSEL, PENABLE, PADDR, PWRITE=0
APB Slave (or APB Slave CDC)
    ↓ cmd_valid, cmd_pwrite=0, cmd_paddr
peakrdl_to_cmdrsp Adapter
    ↓ regblk_req, regblk_req_is_wr=0, regblk_addr
hpet_regs (PeakRDL)
    ← hwif_in (live counter, status)
    ↓ regblk_rd_data
peakrdl_to_cmdrsp Adapter
    ↓ rsp_prdata
APB Slave (or APB Slave CDC)
    ↓ PRDATA
APB Master
```

1.0.6.4.3 Timer Fire Flow

```
hpet_core
    ← Counter increments
    -> Comparator match detected
    -> timer_fired[i] asserts
    -> timer_irq[i] asserts
        ↓
hpet_config_regs
    -> hwif_in.HPET_STATUS.timer_int_status (edge pulse)
        ↓
hpet_regs (PeakRDL)
    -> STATUS register bit latches (sticky)
        ↓
Software reads HPET_STATUS
Software writes W1C to clear
    ↓
hpet_config_regs
    -> timer_int_clear[i] asserts
        ↓
hpet_core
    -> timer_fired[i] clears
    -> timer_irq[i] deasserts
```

1.0.6.5 Clock Domain Organization

1.0.6.5.1 Synchronous Mode (CDC_ENABLE=0)

```
APB Clock Domain (pclk)
+-- apb_slave
+-- hpet_config_regs
+-- hpet_regs
+-- hpet_core

All modules use pclk
No clock domain crossing required
```

1.0.6.5.2 Asynchronous Mode (CDC_ENABLE=1)

```
APB Clock Domain (pclk)
+-- apb_slave_cdc (pclk side)
+-- [CDC boundary]

HPET Clock Domain (hpet_clk)
+-- apb_slave_cdc (hpet_clk side)
+-- hpet_config_regs
+-- hpet_regs
+-- hpet_core

CDC synchronization between pclk and hpet_clk
```

1.0.6.6 Module Communication

1.0.6.6.1 hpet_config_regs -> hpet_core Interface

Control Signals (hpet_config_regs -> hpet_core):

```
output logic                     hpet_enable;           // Global
enable
output logic                     counter_write;         // Counter
write strobe
output logic [63:0]              counter_wdata;         // Counter
write data
output logic [NUM_TIMERS-1:0]    timer_enable;          // Per-timer
enable
output logic [NUM_TIMERS-1:0]    timer_int_enable;      // Per-timer
interrupt enable
output logic [NUM_TIMERS-1:0]    timer_type;            // Per-timer
mode (0=one-shot, 1=periodic)
output logic [NUM_TIMERS-1:0]    timer_size;            // Per-timer
size (0=32-bit, 1=64-bit)
output logic [NUM_TIMERS-1:0]    timer_comp_write;      // Per-timer
comparator write strobe
output logic [63:0]              timer_comp_wdata[NUM_TIMERS];  // Per-
timer data buses
```

Status Signals (hpet_core -> hpet_config_regs):

```
input  logic [63:0]              counter_rdata;         // Live
counter value
input  logic [NUM_TIMERS-1:0]    timer_int_status;      // Per-timer
fire status
```

Interrupt Clearing (hpet_config_regs -> hpet_core):

```
output logic [NUM_TIMERS-1:0]    timer_int_clear;       // Clear fire
flags
```

1.0.6.6.2 hpet_config_regs -> hpet_regs Interface

Uses PeakRDL-generated structs:

```
// From config regs to PeakRDL
input  hpet_regs_pkg::hpet_regs__in_t  hwif_in;

// From PeakRDL to config regs
output hpet_regs_pkg::hpet_regs__out_t hwif_out;
```

1.0.6.7 Resource Allocation

Per-Configuration Estimates (Post-Synthesis):

Component	NUM_TIMERS=2	NUM_TIMERS=3	NUM_TIMERS=8
hpet_core			
- Main counter	64 FF, 70 LUTs	(same)	(same)
- Per-timer logic	256 FF, 170 LUTs	384 FF, 255 LUTs	1024 FF, 680 LUTs
- Subtotal	320 FF, 240 LUTs	448 FF, 325 LUTs	1088 FF, 750 LUTs
hpet_config_regs			
- Mapping logic	~50 FF, ~100 LUTs	~75 FF, ~150 LUTs	~150 FF, ~300 LUTs
- Edge detect	~10 FF, ~20 LUTs	~15 FF, ~30 LUTs	~30 FF, ~60 LUTs
- Subtotal	60 FF, 120 LUTs	90 FF, 180 LUTs	180 FF, 360 LUTs
hpet_regs			
- Register storage	~128 FF, ~100 LUTs	~160 FF, ~125 LUTs	~256 FF, ~200 LUTs
apb_slave (no CDC)			
- APB protocol	~20 FF, ~50 LUTs	(same)	(same)
apb_slave_cdc (with CDC)			
- CDC logic	~100 FF, ~150 LUTs	(same)	(same)
Total (no CDC)	~528 FF, ~510 LUTs	~718 FF, ~680 LUTs	~1544 FF, ~1360 LUTs
Total (with CDC)	~608 FF, ~610 LUTs	~798 FF, ~780 LUTs	~1624 FF, ~1460 LUTs

Scaling: Resource usage is primarily driven by `NUM_TIMERS` parameter. Each additional timer adds ~128 FF and ~85 LUTs.

1.0.6.8 Integration Checklist

When integrating APB HPET:

1. Parameter Selection: - [ ] `NUM_TIMERS`: 2, 3, or 8 timers - [ ] `VENDOR_ID`: 16-bit vendor identification - [ ] `REVISION_ID`: 16-bit revision identification - [ ] `CDC_ENABLE`: 0 for synchronous, 1 for asynchronous clocks

2. Clock Configuration: - [ ] Connect `pclk` (APB clock domain) - [ ] Connect `hpet_clk` (timer clock domain) - [ ] If `CDC_ENABLE=0`: Ensure `pclk = hpet_clk` - [ ] If `CDC_ENABLE=1`: Clocks can be asynchronous

3. Reset Coordination: - [ ] Assert `presetn` (APB reset, active-low) - [ ] Assert `hpet_rst_n` (HPET reset, active-low) - [ ] If `CDC_ENABLE=1`: Ensure both resets overlap at power-on - [ ] Hold resets for >=10 clock cycles

4. APB Interface: - [ ] Connect all APB signals (PSEL, PENABLE, PADDR, etc.) - [ ] PADDR width = 12 bits (supports up to 4KB address space) - [ ] PWDATA/PRDATA width = 32 bits (fixed)

5. Interrupt Outputs: - [ ] Connect `timer_irq[NUM_TIMERS-1:0]` to interrupt controller - [ ] Each timer has independent interrupt output - [ ] Interrupts are active-high, level-sensitive

6. Verification: - [ ] Test register access via APB - [ ] Verify timer operation (one-shot and periodic modes) - [ ] Test interrupt generation and clearing - [ ] Validate CDC if enabled

---

Next: Chapter 2.2 - hpet_config_regs

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

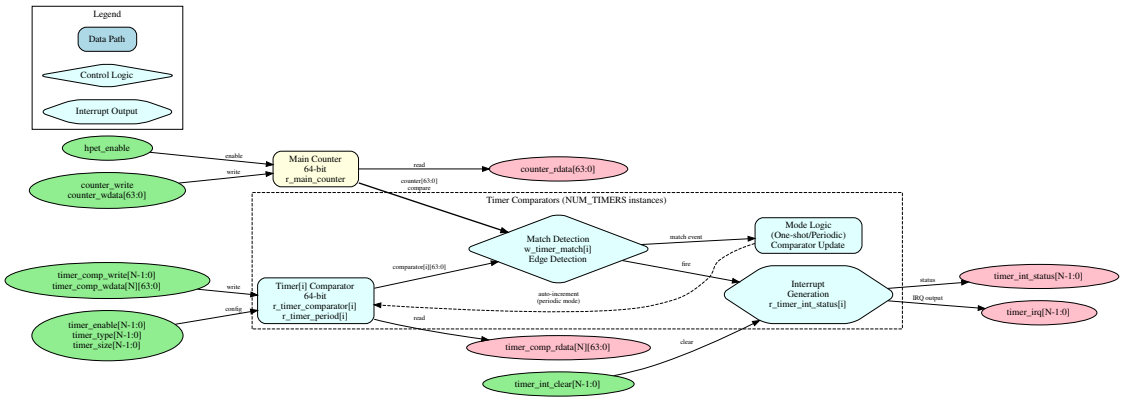
---

1.0.7 HPET Core - Timer Logic

1.0.7.1 Overview

The HPET core (`hpet_core.sv`) implements the fundamental timer functionality: a 64-bit free-running counter, per-timer comparators, and interrupt generation. This module operates entirely in the `hpet_clk` domain and contains all timing-critical logic.

Block Diagram:



HPET Core Block Diagram

Figure: HPET Core architecture showing main counter, timer comparators, match detection, and interrupt generation. *Source: assets/graphviz/hpet_core.gv* | *SVG*

1.0.7.2 Key Features

- **64-bit Free-Running Counter**: Increments every HPET clock cycle, provides timestamp base
- **Configurable Timer Array**: 2, 3, or 8 independent timers (compile-time parameter)
- **64-bit Comparators**: Per-timer comparison values with full counter range
- **Dual Operating Modes**: One-shot and periodic modes per timer
- **Automatic Period Reload**: Periodic mode auto-increments comparator after each fire
- **Individual Interrupts**: Separate fire flag and interrupt output per timer
- **Counter Read/Write Access**: Software can read and write counter value via config registers

1.0.7.3 Interface Specification

1.0.7.3.1 Parameters

Parameter	Type	Default	Range	Description
NUM_TIMERS	int	2	2, 3, 8	Number of independent timers in array

1.0.7.3.2 Clock and Reset

Signal Name	Type	Width	Direction	Description
hpet_clk	logic	1	Input	HPET timer clock

Signal Name	Type	Width	Direction	Description
				(counter increment)
hpet_rst_n	logic	1	Input	Active-low asynchronous reset

1.0.7.3.3 Configuration Interface (from hpet_config_regs)

Signal Name	Type	Width	Direction	Description
hpet_enable	logic	1	Input	Global HPET enable (from HPET_CONFIG[0])
counter_write_enable	logic	1	Input	Write strobe for counter
counter_write_data	logic	64	Input	New counter value (from HPET_COUNTER_LO/HI)
timer_enable[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Input	Per-timer enable (from TIMER_CONFIG[0])
timer_int_enable[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Input	Per-timer interrupt enable (from TIMER_CONFIG[1])
timer_type[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Input	Per-timer mode: 0=One-shot, 1=Periodic
timer_comparator_wr[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Input	Per-timer comparator write strobe
timer_comparator_data[NUM_TIMERS-1:0]	logic [63:0]	NUM_TIMERS×64	Input	Per-timer comparator write data

1.0.7.3.4 Status Interface (to hpet_config_regs)

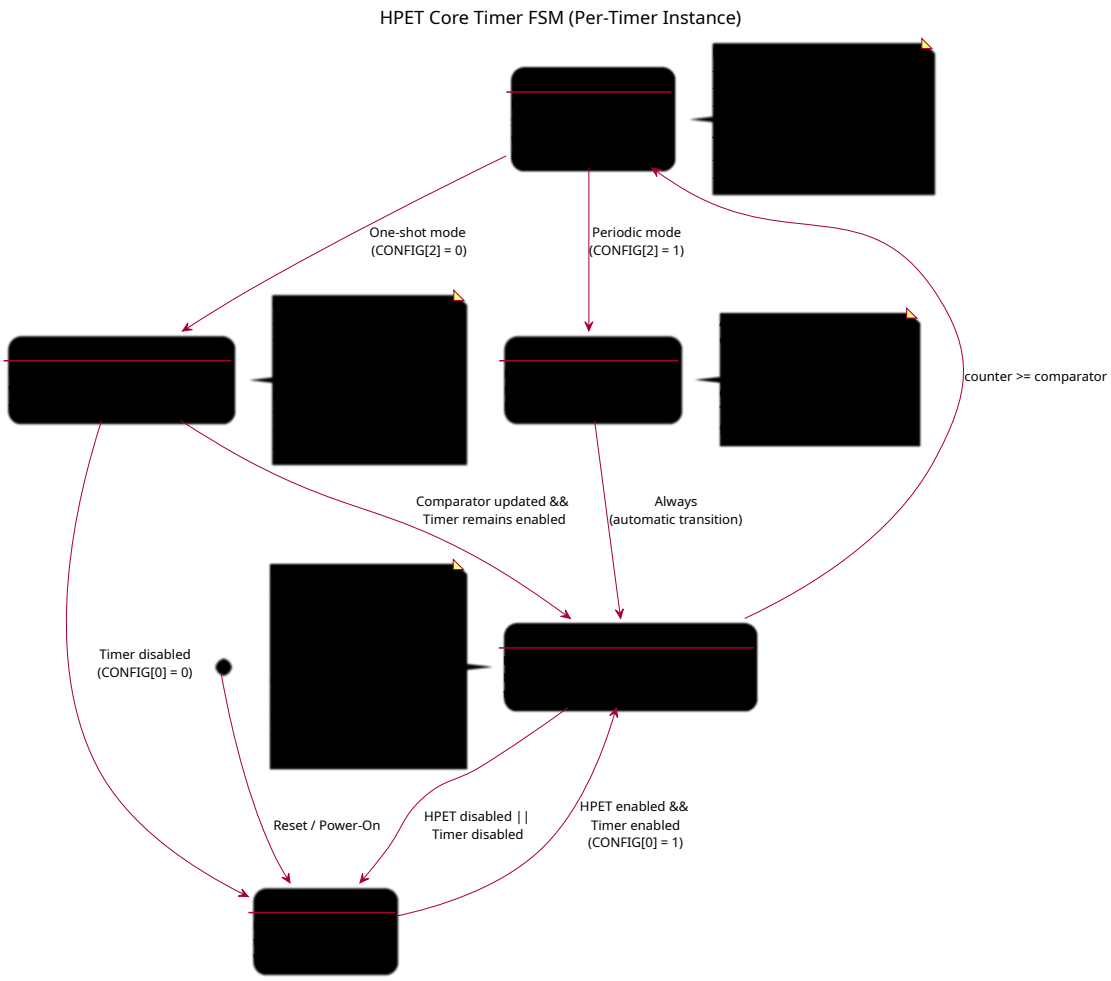
Signal Name	Type	Width	Direction	Description
counter_value	logic	64	Output	Current main counter value (to HPET_COUNTER_LO/HI)
timer_fired[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Output	Per-timer fire flags (to HPET_STATUS)

1.0.7.3.5 Interrupt Interface

Signal Name	Type	Width	Direction	Description
timer_irq[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Output	Per-timer interrupt outputs (active-high)

1.0.7.4 Per-Timer State Machine

Each timer instance implements an identical FSM controlling its operation:



Timer FSM

1.0.7.4.1 FSM States

State	Encoding	Description
IDLE	Default	Timer disabled, waiting for enable signal
ARMED	Active	Timer enabled, monitoring counter vs comparator
FIRE	Transient	Timer match detected, asserting interrupt (1 cycle)
PERIODIC_RELOAD	Transient	Periodic mode: auto-increment comparator (1 cycle)
ONE_SHOT_COMPLETE	Sticky	One-shot mode: timer complete, waiting for reconfigure

Note: FSM is **conceptual** - implementation uses combinational logic rather than explicit state registers for simplicity and timing.

1.0.7.4.2 State Transitions

IDLE -> ARMED: - Condition: `hpet_enable && timer_enable[i]` - Action: Latch current comparator value - Duration: Immediate (next clock cycle)

ARMED -> FIRE: - Condition: `counter_value >= timer_comparator[i]` - Action: Assert `timer_fired[i]` flag - Duration: 1 clock cycle (fire is edge-detected)

FIRE -> PERIODIC_RELOAD: - Condition: `timer_type[i] == 1` (periodic mode) - Action: `timer_comparator[i] <= timer_comparator[i] + timer_period[i]` - Duration: 1 clock cycle

FIRE -> ONE_SHOT_COMPLETE: - Condition: `timer_type[i] == 0` (one-shot mode) - Action: Hold `timer_fired[i]` flag until software clears - Duration: Until STATUS cleared or timer disabled

PERIODIC_RELOAD -> ARMED: - Condition: Always (automatic) - Action: Resume monitoring with new comparator value - Duration: Immediate

ONE_SHOT_COMPLETE -> ARMED: - Condition: Comparator updated while timer remains enabled - Action: Resume monitoring with new comparator value - Duration: Immediate on comparator write strobe

ARMED/ONE_SHOT_COMPLETE -> IDLE: - Condition: `!hpet_enable || !timer_enable[i]` - Action: Clear timer state, stop monitoring - Duration: Immediate

1.0.7.5 Main Counter Logic

1.0.7.5.1 Counter Increment

```
// 64-bit free-running counter
logic [63:0] r_main_counter;

always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n) begin
        r_main_counter <= 64'h0;
    end else if (counter_write_enable) begin
        // Software write to counter
        r_main_counter <= counter_write_data;
    end else if (hpet_enable) begin
        // Continuous increment when HPET enabled
        r_main_counter <= r_main_counter + 64'h1;
    end
    // else: Hold value when HPET disabled
end
```

```
// Output current counter value
assign counter_value = r_main_counter;
```

Key Behavior: - **Reset**: Counter initializes to 0 - **Software Write**: Counter can be written via HPET_COUNTER_LO/HI registers - **Increment**: Counter increments every clock when `hpet_enable = 1` - **Overflow**: Counter wraps from 64'hFFFF_FFFF_FFFF_FFFF to 64'h0 naturally

1.0.7.5.2 Counter Timing

```
Clock:      --+ +-+ +-+ +-+ +-+ +-+ +-
hpet_clk      +-+ +-+ +-+ +-+ +-

Enable:     --------+
hpet_enable         +-------------

Counter:    [N] [N] [N+1][N+2][N+3]
r_main_counter

Latency: 1 cycle from enable to first increment
```

1.0.7.6 Timer Comparator Logic

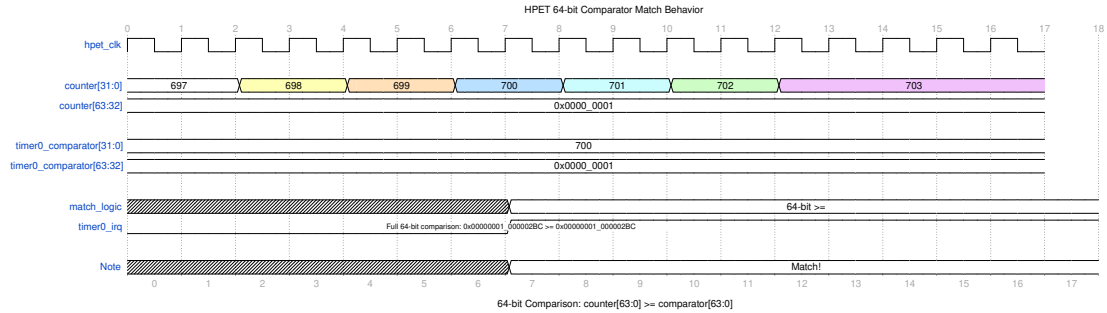
1.0.7.6.1 Comparator Storage (Per-Timer)

```
// Per-timer comparator and period storage
logic [63:0] r_timer_comparator [NUM_TIMERS-1:0];
logic [63:0] r_timer_period [NUM_TIMERS-1:0];

for (genvar i = 0; i < NUM_TIMERS; i++) begin : gen_timer_comparators
    always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
        if (!hpet_rst_n) begin
            r_timer_comparator[i] <= 64'h0;
            r_timer_period[i] <= 64'h0;
        end else if (timer_comparator_wr[i]) begin
            // Software write to comparator
            r_timer_comparator[i] <= timer_comparator_data[i];
            r_timer_period[i] <= timer_comparator_data[i];  // Store
initial period
        end else if (timer_fired[i] && timer_type[i]) begin
            // Periodic mode auto-reload
            r_timer_comparator[i] <= r_timer_comparator[i] +
r_timer_period[i];
        end
        // else: Hold value
    end
end
```

Key Behavior: - **Reset**: Comparator and period clear to 0 - **Initial Write**: Both comparator and period latched from same write - **Periodic Mode**: Comparator auto-increments by period value on each fire - **One-Shot Mode**: Comparator remains constant after initial write

1.0.7.6.2 Match Detection

64-bit Comparator Match Waveform:



Comparator Match Behavior

Use WaveDrom Editor to view/edit, or generate SVG with `wavedrom-cli`

```
// Per-timer match detection (combinational)
logic [NUM_TIMERS-1:0] w_timer_match;

for (genvar i = 0; i < NUM_TIMERS; i++) begin : gen_timer_match
    assign w_timer_match[i] = (r_main_counter >=
r_timer_comparator[i]) &&
                              timer_enable[i] &&
                              hpet_enable;
end
```

Match Conditions: - Counter value >= comparator value - Timer individually enabled (`timer_enable[i] = 1`) - HPET globally enabled (`hpet_enable = 1`)

1.0.7.7 Timer Fire Logic

1.0.7.7.1 Fire Detection (Rising Edge)

```
// Per-timer previous match state for edge detection
logic [NUM_TIMERS-1:0] r_timer_match_prev;

always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n) begin
        r_timer_match_prev <= '0;
    end else begin
        r_timer_match_prev <= w_timer_match;
    end
end
```

```
// Rising edge detection: fire on transition from no-match to match
logic [NUM_TIMERS-1:0] w_timer_fire_edge;

for (genvar i = 0; i < NUM_TIMERS; i++) begin : gen_timer_fire_edge
    assign w_timer_fire_edge[i] = w_timer_match[i] && !
r_timer_match_prev[i];
end
```

Fire Edge Timing:

```
Clock:      --+ +-+ +-+ +-+ +-+ +-
hpet_clk      +-+ +-+ +-+ +-+ +-

Counter:    [99][100][101][102][103]
r_main_counter

Comparator:      [100]
                (constant)

Match:      ------+
w_timer_match     +-----------

Match Prev: --------+
r_timer_match_prev  +---------

Fire Edge:  ----+ +-
w_timer_fire_edge +-

Fired Flag: ----+
timer_fired[i]  +-------------

Note: Fire edge is 1-cycle pulse on rising edge of match
```

1.0.7.7.2 Fire Flag Management

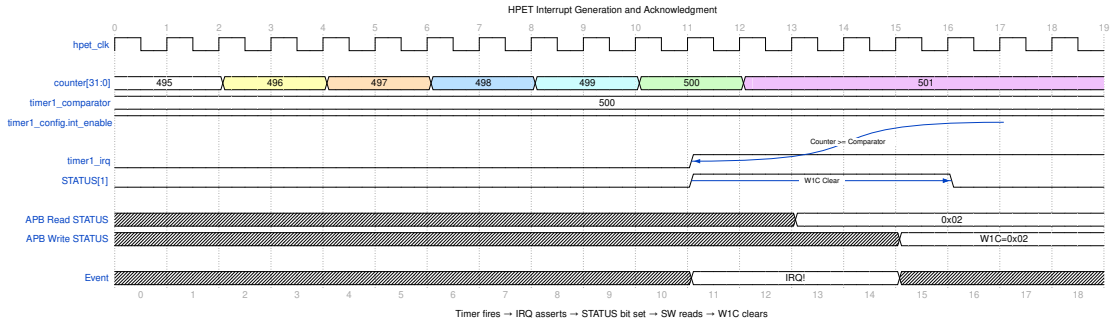
```
// Per-timer fired flag (sticky in one-shot mode, pulse in periodic
mode)
logic [NUM_TIMERS-1:0] r_timer_fired;

for (genvar i = 0; i < NUM_TIMERS; i++) begin : gen_timer_fired
    always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
        if (!hpet_rst_n || !timer_enable[i]) begin
            r_timer_fired[i] <= 1'b0;
        end else if (w_timer_fire_edge[i]) begin
            r_timer_fired[i] <= 1'b1;  // Set on fire edge
        end else if (!timer_type[i]) begin
```

```
            // One-shot mode: hold fired flag until software clears
STATUS
            r_timer_fired[i] <= r_timer_fired[i];  // Sticky
        end else begin
            // Periodic mode: clear after 1 cycle (pulse)
            r_timer_fired[i] <= 1'b0;
        end
    end
end

// Output fire flags to config regs (connect to HPET_STATUS)
assign timer_fired = r_timer_fired;
```

Fire Flag Behavior: - **One-Shot Mode**: Sticky (remains 1 until STATUS cleared by software) - **Periodic Mode**: Pulse (1 cycle per fire, auto-clears)

1.0.7.8 Interrupt Generation

Interrupt Generation and Acknowledgment Waveform:



Interrupt Generation

Use WaveDrom Editor to view/edit, or generate SVG with `wavedrom-cli`

1.0.7.8.1 Interrupt Output Logic

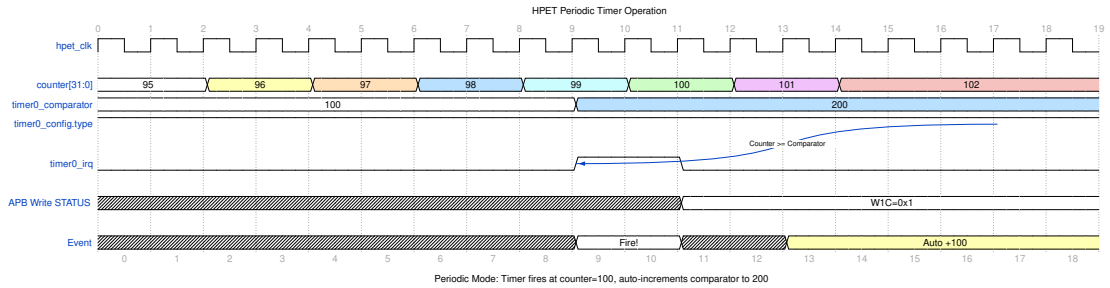
```
// Per-timer interrupt output (combinational, follows STATUS register)
for (genvar i = 0; i < NUM_TIMERS; i++) begin : gen_timer_irq
    assign timer_irq[i] = timer_fired[i] && timer_int_enable[i];
end
```

Interrupt Behavior: - **Combinational**: Interrupt follows fire flag (no additional latency) - **Maskable**: `timer_int_enable[i]` from TIMER_CONFIG[1] gates interrupt - **Sticky (One-Shot)**: Interrupt remains asserted until STATUS cleared - **Pulse (Periodic)**: Interrupt pulses on each fire event

Interrupt Clearing: Software clears interrupts by writing 1 to corresponding HPET_STATUS bit (W1C). The `timer_fired` flag is managed in `hpet_config_regs` wrapper, not in hpet_core.

1.0.7.9 Periodic Mode Details

Periodic Timer Waveform:



Periodic Timer Operation

Use WaveDrom Editor to view/edit, or generate SVG with `wavedrom-cli`

1.0.7.9.1 Period Storage and Auto-Reload

Initial Comparator Write:

```
Software writes: TIMER0_COMPARATOR = 1000
Result:
  r_timer_comparator[0] = 1000
  r_timer_period[0] = 1000  (also latched)
```

First Fire (at counter = 1000):

```
Fire edge detected
-> timer_fired[0] asserts
-> Comparator auto-reloads:
  r_timer_comparator[0] = 1000 + 1000 = 2000
```

Second Fire (at counter = 2000):

```
Fire edge detected
-> timer_fired[0] asserts
-> Comparator auto-reloads:
  r_timer_comparator[0] = 2000 + 1000 = 3000
```

Process repeats indefinitely until timer disabled

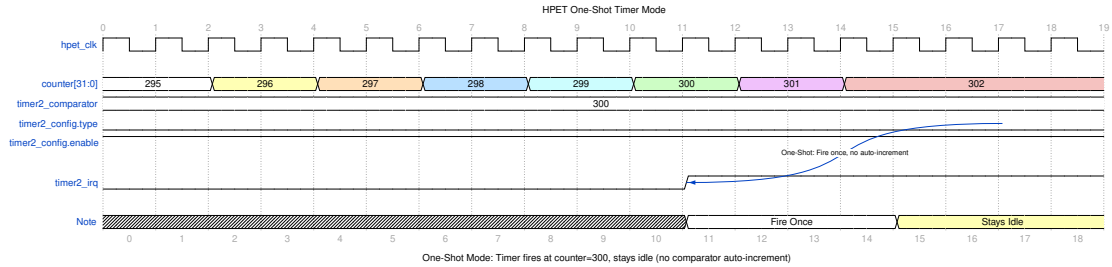
1.0.7.9.2 Periodic Mode Timing Example

```
Clock Cycles:   0   1000 1001 2000 2001 3000 3001 ...

Counter:        0 -> 1000 1001 2000 2001 3000 3001 ...

Comparator:     [1000] [2000] [3000] [4000] ...
                   ↑       ↑       ↑
```

```
                Fire 1  Fire 2  Fire 3

timer_fired:    --+ +-+ +-+ +-...
                  +-+ +-+ +-

timer_irq:      --+ +-+ +-+ +-...
                  +-+ +-+ +-

Period = 1000 HPET clock cycles (constant)
```

1.0.7.10 One-Shot Mode Details

One-Shot Timer Waveform:



One-Shot Mode Operation

Use WaveDrom Editor to view/edit, or generate SVG with `wavedrom-cli`

1.0.7.10.1 Fire-Once Behavior

Initial Comparator Write:

```
Software writes: TIMER0_COMPARATOR = 5000
Result:
  r_timer_comparator[0] = 5000
  (period not used in one-shot mode)
```

Fire Event (at counter = 5000):

```
Fire edge detected
-> timer_fired[0] asserts (sticky)
-> Comparator remains at 5000 (no auto-reload)
-> Interrupt remains asserted
```

Interrupt Clearing:

```
Software writes: HPET_STATUS[0] = 1 (W1C)
Result:
  timer_fired[0] clears
  timer_irq[0] clears
```

Reconfiguration:

```
Software writes: TIMER0_COMPARATOR = 10000
Result:
  r_timer_comparator[0] = 10000
  Timer re-arms, waits for counter = 10000
```

1.0.7.10.2 One-Shot Mode Timing Example

```
Clock Cycles:   0   5000 5001 5002 ...

Counter:        0 -> 5000 5001 5002 ...

Comparator:     [5000] [5000] [5000] ...
                   ↑
                Fire (once)

timer_fired:    --+
                  +--------------... (sticky until SW clear)

timer_irq:      --+
                  +--------------... (follows fired flag)

Software Write: ------+ +-
HPET_STATUS[0]=1      +-

timer_fired:    --+     +-
(after clear)     +-----+

Fire only once, interrupt sticky until software clear
```

1.0.7.11 Resource Utilization

Per-Timer Resources (Estimated): - 64-bit comparator register: 64 flip-flops - 64-bit period register: 64 flip-flops - Match comparator: 64-bit >= comparison (~80 LUTs) - Fire edge detection: 2 flip-flops + XOR gate - Total per timer: ~128 flip-flops, ~85 LUTs

Shared Resources: - 64-bit main counter: 64 flip-flops + 64-bit adder (~70 LUTs) - Global enable logic: ~10 LUTs

Total (NUM_TIMERS = 3): - Flip-flops: 64 + (128 × 3) = 448 FF - LUTs: 80 + (85 × 3) = 335 LUTs

---

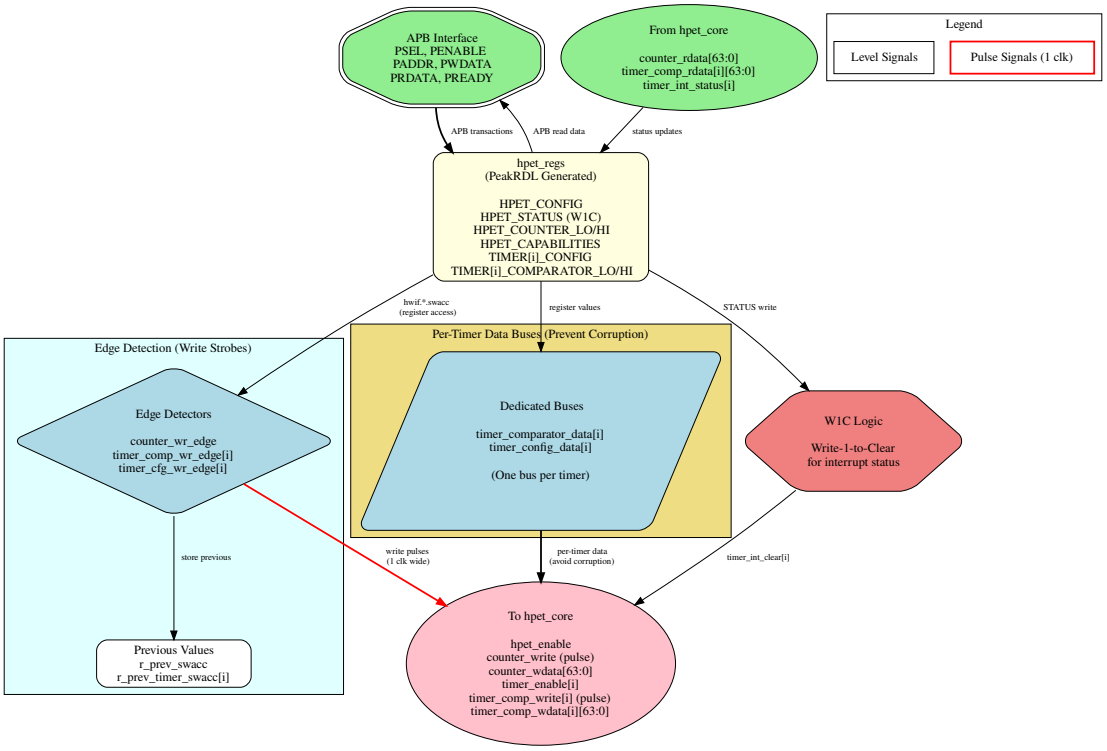
Next: Chapter 2.2 - hpet_config_regs

1.0.8 HPET Configuration Registers - PeakRDL Wrapper

1.0.8.1 Overview

The `hpet_config_regs` module serves as the critical bridge between the PeakRDL-generated register file (`hpet_regs.sv`) and the HPET core timer logic (`hpet_core.sv`). This wrapper handles interface adaptation, per-timer data bus isolation, and register write edge detection.

Block Diagram:



HPET Config Registers Block Diagram

Figure: HPET Config Registers architecture showing APB interface, PeakRDL registers, edge detection, per-timer data buses, and W1C logic. Source: assets/graphviz/hpet_config_regs.gv | SVG

1.0.8.2 Key Responsibilities

1. **PeakRDL Integration:** Instantiates `hpet_regs.sv` and `peakrdl_to_cmdrsp` adapter

2. **Interface Mapping:** Converts PeakRDL hardware interface to HPET core signals
3. **Per-Timer Data Buses:** Implements dedicated 64-bit data paths per timer (prevents corruption)
4. **Edge Detection:** Generates write strobes from register updates
5. **Counter Write Handling:** Captures software writes to counter registers
6. **Interrupt Management:** Handles W1C status clearing and interrupt feedback

1.0.8.3 Interface Specification

1.0.8.3.1 Parameters

Parameter	Type	Default	Range	Description
VENDOR_ID	int	1	0-65535	Vendor identification (read-only in HPET_ID)
REVISION_ID	int	1	0-65535	Revision identification (read-only in HPET_ID)
NUM_TIMERS	int	2	2, 3, 8	Number of independent timers in array

1.0.8.3.2 Clock and Reset

Signal Name	Type	Width	Direction	Description
clk	logic	1	Input	Configuration clock (pclk or hpet_clk based on CDC_ENABLE)
rst_n	logic	1	Input	Active-low asynchronous reset

1.0.8.3.3 Command/Response Interface (from APB Slave)

Signal Name	Type	Width	Direction	Description
cmd_valid	logic	1	Input	Command valid
cmd_ready	logic	1	Output	Command

Signal Name	Type	Width	Direction	Description
				ready
cmd_pwrite	logic	1	Input	Command write (1) or read (0)
cmd_paddr	logic	12	Input	Command address
cmd_pwdata	logic	32	Input	Command write data
cmd_pstrb	logic	4	Input	Command write byte strobes
rsp_valid	logic	1	Output	Response valid
rsp_ready	logic	1	Input	Response ready
rsp_prdata	logic	32	Output	Response read data
rsp_pslverr	logic	1	Output	Response error flag

1.0.8.3.4 HPET Core Interface (to hpet_core.sv)

Global Configuration: | Signal Name | Type | Width | Direction | Description | |————-|——|——-|———|————-| | **hpet_enable** | logic | 1 | Output | Global HPET enable (from HPET_CONFIG[0]) | | **legacy_replacement** | logic | 1 | Output | Legacy replacement mode (from HPET_CONFIG[1]) |

Counter Interface: | Signal Name | Type | Width | Direction | Description | |————-|——|——-|———|————-| | **counter_write** | logic | 1 | Output | Counter write strobe (pulse) | | **counter_wdata** | logic | 64 | Output | Counter write data (combined LO/HI) | | **counter_rdata** | logic | 64 | Input | Live counter value (from hpet_core) |

Per-Timer Configuration: | Signal Name | Type | Width | Direction | Description | |————-|——|——-|———|————-| | **timer_enable[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer enable bits (from TIMER_CONFIG[2]) | | **timer_int_enable[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer interrupt enable (from TIMER_CONFIG[3]) | | **timer_type[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer mode: 0=One-shot, 1=Periodic (from TIMER_CONFIG[4]) | | **timer_size[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer size: 0=32-bit, 1=64-bit (from TIMER_CONFIG[5]) | |

timer_value_set[NUM_TIMERS-1:0] | logic | NUM_TIMERS | Output | Per-timer accumulator mode (from TIMER_CONFIG[6]) |

Per-Timer Comparator (Dedicated Buses): | Signal Name | Type | Width | Direction | Description | |————-|——|——-|————|————-| | **timer_comp_write[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer comparator write strobes | | **timer_comp_wdata[NUM_TIMERS]** | logic [63:0] | NUM_TIMERS×64 | Output | Per-timer comparator data (LO/HI combined) | | **timer_comp_write_high** | logic | 1 | Output | High half write detection | | **timer_comp_rdata[NUM_TIMERS]** | logic [63:0] | NUM_TIMERS×64 | Input | Per-timer comparator read data |

Interrupt Status: | Signal Name | Type | Width | Direction | Description | |————-|——|——-|————|————-| | **timer_int_status[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Input | Per-timer fire status (from hpet_core) | | **timer_int_clear[NUM_TIMERS-1:0]** | logic | NUM_TIMERS | Output | Per-timer status clear (W1C pulse) |

1.0.8.4 Internal Architecture

1.0.8.4.1 Component Instantiation

1. Protocol Adapter:

```
peakrdl_to_cmdrsp #(
    .ADDR_WIDTH(12),
    .DATA_WIDTH(32)
) u_adapter (
    .aclk(clk), .aresetn(rst_n),
    // cmd/rsp interface (external)
    .cmd_valid, .cmd_ready, .cmd_pwrite, .cmd_paddr, .cmd_pwdata, .cmd
_pstrb,
    .rsp_valid, .rsp_ready, .rsp_prdata, .rsp_pslverr,
    // PeakRDL passthrough interface (to register block)
    .regblk_req, .regblk_req_is_wr, .regblk_addr, .regblk_wr_data, .re
gblk_wr_biten,
    .regblk_req_stall_wr, .regblk_req_stall_rd,
    .regblk_rd_ack, .regblk_rd_err, .regblk_rd_data,
    .regblk_wr_ack, .regblk_wr_err
);
```

2. PeakRDL Register Block:

```
hpet_regs u_hpet_regs (
    .clk(clk),
    .rst(~rst_n),  // PeakRDL uses active-high reset
    // Passthrough CPU interface
    .s_cpuif_req(regblk_req),
    .s_cpuif_req_is_wr(regblk_req_is_wr),
    .s_cpuif_addr(regblk_addr[8:0]),  // 9-bit internal addressing
    .s_cpuif_wr_data(regblk_wr_data),
```

```
    .s_cpuif_wr_biten(regblk_wr_biten),
    .s_cpuif_req_stall_wr(regblk_req_stall_wr),
    .s_cpuif_req_stall_rd(regblk_req_stall_rd),
    .s_cpuif_rd_ack(regblk_rd_ack),
    .s_cpuif_rd_err(regblk_rd_err),
    .s_cpuif_rd_data(regblk_rd_data),
    .s_cpuif_wr_ack(regblk_wr_ack),
    .s_cpuif_wr_err(regblk_wr_err),
    // Hardware interface
    .hwif_in(hwif_in),
    .hwif_out(hwif_out)
);
```

1.0.8.5 Mapping Logic Details

1.0.8.5.1 Global Configuration Mapping

Direct assignment from PeakRDL outputs:

```
assign hpet_enable = hwif_out.HPET_CONFIG.hpet_enable.value;
assign legacy_replacement =
hwif_out.HPET_CONFIG.legacy_replacement.value;
```

1.0.8.5.2 Counter Write Detection

Uses address-based detection and data capture:

```
// Detect which register was written
assign counter_lo_written = regblk_req && regblk_req_is_wr &&
(regblk_addr[8:0] == 9'h010);
assign counter_hi_written = regblk_req && regblk_req_is_wr &&
(regblk_addr[8:0] == 9'h014);

// Capture software-written values from write data bus
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        last_sw_counter_lo <= '0;
        last_sw_counter_hi <= '0;
    end else begin
        if (counter_lo_written) last_sw_counter_lo <= regblk_wr_data;
        if (counter_hi_written) last_sw_counter_hi <= regblk_wr_data;
    end
end

// Counter write strobe asserted when software modifies either half
assign counter_write = hwif_out.HPET_COUNTER_LO.counter_lo.swmod ||
                       hwif_out.HPET_COUNTER_HI.counter_hi.swmod;
```

```
// Combined 64-bit write data
assign counter_wdata = {last_sw_counter_hi, last_sw_counter_lo};
```

Timing:

```
Clock:          -+ +-+ +-+ +-+ +-+ +-
clk             +-+ +-+ +-+ +-

Write:          ---+ +---------
counter_lo_written+-

Data:           [OLD][NEW][NEW]
regblk_wr_data

Captured:       [OLD][OLD][NEW]
last_sw_counter_lo

swmod:          ----+ +-----
                +-

counter_write:----+ +-----
                +-

Note: 1-cycle pulse when software writes
```

1.0.8.5.3 Timer Configuration Mapping

Per-timer array mapping:

```
generate
    for (genvar i = 0; i < NUM_TIMERS; i++) begin : g_timer_mapping
        assign timer_enable[i]     =
hwif_out.TIMER[i].TIMER_CONFIG.timer_enable.value;
        assign timer_int_enable[i] =
hwif_out.TIMER[i].TIMER_CONFIG.timer_int_enable.value;
        assign timer_type[i]       =
hwif_out.TIMER[i].TIMER_CONFIG.timer_type.value;
        assign timer_size[i]       =
hwif_out.TIMER[i].TIMER_CONFIG.timer_size.value;
        assign timer_value_set[i]  =
hwif_out.TIMER[i].TIMER_CONFIG.timer_value_set.value;
    end
endgenerate
```

1.0.8.5.4 Per-Timer Data Bus Architecture (Corruption Fix)

The Problem: Early designs shared a single 64-bit bus for all timer comparators. Rapid writes to different timers caused corruption when one timer's data overwrote another timer's registers.

The Solution: Each timer gets a dedicated 64-bit data bus, preventing any possibility of cross-timer corruption:

```
// ✓ CORRECT: Per-timer dedicated data buses
generate
    for (genvar i = 0; i < NUM_TIMERS; i++) begin : g_timer_wdata
        assign timer_comp_wdata[i] = {
            hwif_out.TIMER[i].TIMER_COMPARATOR_HI.timer_comp_hi.value,
            hwif_out.TIMER[i].TIMER_COMPARATOR_LO.timer_comp_lo.value
        };
    end
endgenerate

// Per-timer write strobe generation (edge detection)
generate
    for (genvar i = 0; i < NUM_TIMERS; i++) begin : g_timer_wr_detect
        always_ff @(posedge clk or negedge rst_n) begin
            if (!rst_n) begin
                prev_timer_comp_lo[i] <= '0;
                prev_timer_comp_hi[i] <= '0;
            end else begin
                prev_timer_comp_lo[i] <=
hwif_out.TIMER[i].TIMER_COMPARATOR_LO.timer_comp_lo.value;
                prev_timer_comp_hi[i] <=
hwif_out.TIMER[i].TIMER_COMPARATOR_HI.timer_comp_hi.value;
            end
        end

        assign timer_comp_write[i] =
            (hwif_out.TIMER[i].TIMER_COMPARATOR_LO.timer_comp_lo.value
!= prev_timer_comp_lo[i]) ||
            (hwif_out.TIMER[i].TIMER_COMPARATOR_HI.timer_comp_hi.value
!= prev_timer_comp_hi[i]);
    end
endgenerate
```

Architecture Benefit:

```
Timer 0:  hwif.TIMER[0].COMP_LO/HI -> timer_comp_wdata[0] -> hpet_core
timer 0 ONLY
Timer 1:  hwif.TIMER[1].COMP_LO/HI -> timer_comp_wdata[1] -> hpet_core
timer 1 ONLY
Timer 2:  hwif.TIMER[2].COMP_LO/HI -> timer_comp_wdata[2] -> hpet_core
timer 2 ONLY

No shared bus -> No corruption possible
```

1.0.8.5.5 Interrupt Status Handling

Edge Detection for Sticky Interrupts:

PeakRDL sticky interrupt fields expect edge pulses (not levels). The wrapper implements edge detection:

```
// Previous state storage
logic [NUM_TIMERS-1:0] prev_timer_int_status;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        prev_timer_int_status <= '0;
    end else begin
        prev_timer_int_status <= timer_int_status;
    end
end

// Detect rising edge (0->1 transition)
assign timer_int_rising_edge = timer_int_status &
~prev_timer_int_status;

// Feed edge-detected pulse to PeakRDL hwset
assign hwif_in.HPET_STATUS.timer_int_status.hwset = |
timer_int_rising_edge;

// Feed current level to next (for multi-bit sticky logic)
assign hwif_in.HPET_STATUS.timer_int_status.next = {{(8-NUM_TIMERS)
{1'b0}}, timer_int_status};
```

Interrupt Clearing (W1C):

When software writes 1 to HPET_STATUS bit to clear (W1C), the wrapper generates a clear pulse to hpet_core:

```
// Detect when software writes W1C to HPET_STATUS
// PeakRDL swmod signal pulses when SW modifies the field
assign timer_int_clear =
{NUM_TIMERS{hwif_out.HPET_STATUS.timer_int_status.swmod}} &
timer_int_status;
```

Timing:

```
Clock:          -+ +-+ +-+ +-
clk             +-+ +-+ +-

Timer Fires:    --+ +-------
timer_int_status  +-
```

```
Edge Detect:     ----+ +-----
timer_int_rising_edge+-

hwset Pulse:     ----+ +-----
hwif_in.hwset    +-

PeakRDL Sticky:  --+
STATUS bit       +---------

SW Write W1C:    --------+ +-
swmod pulse              +-

Clear Pulse:     --------+ +-
timer_int_clear          +-

Timer Clears:    --+       +-
timer_int_status   +-------+

Note: Edge detection + W1C clearing flow
```

1.0.8.6 Register-to-Core Signal Summary

Critical Signals:

1. **hpet_enable:** Level signal, directly gates counter incrementing
2. **counter_write:** Pulse (1 cycle) when software writes counter
3. **counter_wdata:** Captured value from software write
4. **timer_enable[i]:** Level signal per timer
5. **timer_comp_write[i]:** Pulse (1 cycle) when software writes comparator
6. **timer_comp_wdata[i]:** Per-timer dedicated data bus (corruption-proof)
7. **timer_int_clear[i]:** Pulse (1 cycle) when software clears status W1C

Signal Types: - **Level Signals:** Direct PeakRDL `.value` outputs (enable, type, size) - **Pulse Signals:** Edge-detected from register changes (write strobes, clears) - **Data Buses:** Captured or combined register values (counter, comparators)

1.0.8.7 Resource Utilization

Configuration Register Logic (hpet_config_regs only, excluding hpet_regs):

Component	NUM_TIMERS=2	NUM_TIMERS=3	NUM_TIMERS=8
Mapping Logic	~50 FF, ~100 LUTs	~75 FF, ~150 LUTs	~150 FF, ~300 LUTs

Component	NUM_TIMERS=2	NUM_TIMERS=3	NUM_TIMERS=8
Edge Detect	~10 FF, ~20 LUTs	~15 FF, ~30 LUTs	~30 FF, ~60 LUTs
Interrupt Handling	~10 FF, ~20 LUTs	~15 FF, ~30 LUTs	~30 FF, ~60 LUTs
Total	~70 FF, ~140 LUTs	~105 FF, ~210 LUTs	~210 FF, ~420 LUTs

Scaling: Primarily driven by number of timers. Each additional timer adds ~35 FF and ~70 LUTs for mapping and edge detection logic.

Next: Chapter 2.3 - hpet_regs (PeakRDL)

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

1.0.9 HPET Registers - PeakRDL Generated Register File

1.0.9.1 Overview

The `hpet_regs` module is auto-generated from the SystemRDL specification (`rtl/peakrdl/hpet_regs.rdl`) using the PeakRDL toolchain. It implements the complete HPET register file with proper field access semantics (RO, RW, W1C), hardware interface integration, and CPU interface protocol handling.

Single Source of Truth: All register definitions, addresses, field widths, and access properties are specified in the SystemRDL file. The generated RTL is deterministic and regeneratable.

Generation Command:

```
cd projects/components/apb_hpet/rtl/peakrdl
peakrdl regblock hpet_regs.rdl --cpuif passthrough -o ../
```

Generated Files: - `hpet_regs.sv` - Register implementation - `hpet_regs_pkg.sv` - Package with structs and parameters

1.0.9.2 Module Interface

1.0.9.2.1 Parameters

No user-configurable parameters. All configuration is baked into the generated code from SystemRDL.

Compile-Time Constants (from SystemRDL):

```
localparam VENDOR_ID = 1;          // From RDL: vendor_id field default
localparam REVISION_ID = 1;        // From RDL: revision_id field
default
localparam NUM_TIMERS = 8;         // From RDL: TIMER[0:7] array size
```

Note: These values are fixed at generation time. To change them, modify `hpet_regs.rdl` and regenerate.

1.0.9.2.2 Clock and Reset

Signal Name	Type	Width	Direction	Description
clk	wire	1	Input	Register clock (pclk or hpet_clk based on CDC_ENABLE)
rst	wire	1	Input	**Active-high** reset (PeakRDL convention)

⚠️ **Important:** PeakRDL uses active-high reset. The wrapper (`hpet_config_regs.sv`) inverts `rst_n` before connecting.

1.0.9.2.3 CPU Interface (Passthrough Protocol)

Signal Name	Type	Width	Direction	Description
s_cpuif_req	wire	1	Input	CPU request valid
s_cpuif_req_is_wr	wire	1	Input	Request is write (1) or read (0)
s_cpuif_addr	wire	9	Input	Address (byte-aligned, bits [8:0])
s_cpuif_wr_data	wire	32	Input	Write data
s_cpuif_wr_biten	wire	32	Input	Write byte enable (bit-level)
s_cpuif_req_stall_wr	wire	1	Output	Stall write request (always 0 for HPET)
s_cpuif_req_stall_rd	wire	1	Output	Stall read request (always 0 for HPET)
s_cpuif_rd_ack	wire	1	Output	Read acknowledgment

Signal Name	Type	Width	Direction	Description
s_cpuif_rd_err	wire	1	Output	Read error (decoding error)
s_cpuif_rd_data	wire	32	Output	Read data
s_cpuif_wr_ack	wire	1	Output	Write acknowledgment
s_cpuif_wr_err	wire	1	Output	Write error (always 0 for HPET)

Protocol Characteristics: - **Latency:** 1 cycle for both reads and writes - **Stalls:** Never stall (HPET registers have single-cycle access) - **Errors:** Read error on unmapped address, writes always succeed

1.0.9.2.4 Hardware Interface (Structs)

```
input  hpet_regs_pkg::hpet_regs__in_t  hwif_in;   // From hardware to registers
output hpet_regs_pkg::hpet_regs__out_t hwif_out;  // From registers to hardware
```

Structure Definitions (in hpet_regs_pkg.sv):

The package defines comprehensive structs for all registers and fields. Key excerpts:

```
package hpet_regs_pkg;

    // Hardware input struct (hardware -> registers)
    typedef struct packed {
        struct packed {
            logic [4:0] next;  // num_tim_cap field value
        } num_tim_cap;
    } HPET_ID__in_t;

    typedef struct packed {
        logic [7:0] next;   // Next value for status bits
        logic hwset;        // Hardware set pulse
    } timer_int_status__in_t;

    typedef struct packed {
        logic [31:0] next;  // Next counter value
    } counter_lo__in_t;

    // ... additional field structs ...
```

```
    // Complete input struct
    typedef struct packed {
        HPET_ID__in_t HPET_ID;
        timer_int_status__in_t HPET_STATUS.timer_int_status;
        counter_lo__in_t HPET_COUNTER_LO.counter_lo;
        counter_hi__in_t HPET_COUNTER_HI.counter_hi;
        // ... additional register fields ...
    } hpet_regs__in_t;

    // Hardware output struct (registers -> hardware)
    typedef struct packed {
        struct packed {
            logic value;     // Current field value
        } hpet_enable;
        struct packed {
            logic value;
        } legacy_replacement;
    } HPET_CONFIG__out_t;

    typedef struct packed {
        logic swmod;         // Software modified (write detected)
    } timer_int_status__out_t;

    typedef struct packed {
        logic [31:0] value;  // Current register value
        logic swmod;         // Software modified
    } counter_lo__out_t;

    // ... additional field structs ...

    // Complete output struct
    typedef struct packed {
        HPET_CONFIG__out_t HPET_CONFIG;
        timer_int_status__out_t HPET_STATUS.timer_int_status;
        counter_lo__out_t HPET_COUNTER_LO.counter_lo;
        counter_hi__out_t HPET_COUNTER_HI.counter_hi;
        TIMER__out_t TIMER[7:0];  // Timer array
        // ... additional registers ...
    } hpet_regs__out_t;

endpackage
```

1.0.9.3 Register Implementation

1.0.9.3.1 Address Decoding

PeakRDL generates a decoded register strobe struct:

```
typedef struct {
    logic HPET_ID;
    logic HPET_CONFIG;
    logic HPET_STATUS;
    logic RESERVED_0C;
    logic HPET_COUNTER_LO;
    logic HPET_COUNTER_HI;
    struct {
        logic TIMER_CONFIG;
        logic TIMER_COMPARATOR_LO;
        logic TIMER_COMPARATOR_HI;
        logic RESERVED;
    } TIMER[8];
} decoded_reg_strb_t;

decoded_reg_strb_t decoded_reg_strb;
```

Decoding Logic:

```
always_comb begin
    decoded_reg_strb.HPET_ID = cpuif_req_masked & (cpuif_addr ==
9'h0);
    decoded_reg_strb.HPET_CONFIG = cpuif_req_masked & (cpuif_addr ==
9'h4);
    decoded_reg_strb.HPET_STATUS = cpuif_req_masked & (cpuif_addr ==
9'h8);
    decoded_reg_strb.RESERVED_0C = cpuif_req_masked & (cpuif_addr ==
9'hc);
    decoded_reg_strb.HPET_COUNTER_LO = cpuif_req_masked & (cpuif_addr
== 9'h10);
    decoded_reg_strb.HPET_COUNTER_HI = cpuif_req_masked & (cpuif_addr
== 9'h14);

    for(int i0=0; i0<8; i0++) begin
        decoded_reg_strb.TIMER[i0].TIMER_CONFIG =
            cpuif_req_masked & (cpuif_addr == 9'h100 + (9)'(i0) *
9'h20);
        decoded_reg_strb.TIMER[i0].TIMER_COMPARATOR_LO =
            cpuif_req_masked & (cpuif_addr == 9'h104 + (9)'(i0) *
9'h20);
        decoded_reg_strb.TIMER[i0].TIMER_COMPARATOR_HI =
            cpuif_req_masked & (cpuif_addr == 9'h108 + (9)'(i0) *
9'h20);
        decoded_reg_strb.TIMER[i0].RESERVED =
            cpuif_req_masked & (cpuif_addr == 9'h10c + (9)'(i0) *
9'h20);
    end
end
```

1.0.9.3.2 Field Logic

Each field is implemented with: - **Combo Logic:** Determines next value based on SW write, HW input, or current value - **Sequential Logic:** Stores field value in flip-flops - **Output Assignment:** Drives `hwif_out` struct

Example - HPET_CONFIG.hpet_enable Field:

```
// Field: hpet_regs.HPET_CONFIG.hpet_enable
always_comb begin
    automatic logic [0:0] next_c;
    automatic logic load_next_c;

    next_c = field_storage.HPET_CONFIG.hpet_enable.value;  // Default:
hold
    load_next_c = '0;

    if(decoded_reg_strb.HPET_CONFIG && decoded_req_is_wr) begin  // SW
write
        next_c = (field_storage.HPET_CONFIG.hpet_enable.value &
~decoded_wr_biten[0:0]) |
                 (decoded_wr_data[0:0] & decoded_wr_biten[0:0]);
        load_next_c = '1;
    end

    field_combo.HPET_CONFIG.hpet_enable.next = next_c;
    field_combo.HPET_CONFIG.hpet_enable.load_next = load_next_c;
end

always_ff @(posedge clk) begin
    if(rst) begin
        field_storage.HPET_CONFIG.hpet_enable.value <= 1'h0;  // Reset
value
    end else begin
        if(field_combo.HPET_CONFIG.hpet_enable.load_next) begin
            field_storage.HPET_CONFIG.hpet_enable.value <=
field_combo.HPET_CONFIG.hpet_enable.next;
        end
    end
end

assign hwif_out.HPET_CONFIG.hpet_enable.value =
field_storage.HPET_CONFIG.hpet_enable.value;
```

Example - HPET_STATUS.timer_int_status Field (W1C with HW set):

```
// Field: hpet_regs.HPET_STATUS.timer_int_status
always_comb begin
```

```
    automatic logic [7:0] next_c;
    automatic logic load_next_c;

    next_c = field_storage.HPET_STATUS.timer_int_status.value;
    load_next_c = '0;

    if(decoded_reg_strb.HPET_STATUS && decoded_req_is_wr) begin  // SW
write 1 to clear
        next_c = field_storage.HPET_STATUS.timer_int_status.value &
                ~(decoded_wr_data[7:0] & decoded_wr_biten[7:0]);
        load_next_c = '1;

    end else if((field_storage.HPET_STATUS.timer_int_status.value ==
'0) &&
                (hwif_in.HPET_STATUS.timer_int_status.next != '0))
begin  // Multi-bit sticky
        next_c = hwif_in.HPET_STATUS.timer_int_status.next;
        load_next_c = '1;

    end else if(hwif_in.HPET_STATUS.timer_int_status.hwset) begin  //
HW set
        next_c = '1;
        load_next_c = '1;
    end

    field_combo.HPET_STATUS.timer_int_status.next = next_c;
    field_combo.HPET_STATUS.timer_int_status.load_next = load_next_c;
end

always_ff @(posedge clk) begin
    if(field_combo.HPET_STATUS.timer_int_status.load_next) begin
        field_storage.HPET_STATUS.timer_int_status.value <=
field_combo.HPET_STATUS.timer_int_status.next;
    end
end

// swmod signal: pulsed when software modifies field
assign hwif_out.HPET_STATUS.timer_int_status.swmod =
    decoded_reg_strb.HPET_STATUS && decoded_req_is_wr && |
(decoded_wr_biten[7:0]);
```

Example - HPET_COUNTER_LO Field (HW write with SW precedence):

```
// Field: hpet_regs.HPET_COUNTER_LO.counter_lo
always_comb begin
    automatic logic [31:0] next_c;
    automatic logic load_next_c;
```

```
    next_c = field_storage.HPET_COUNTER_LO.counter_lo.value;
    load_next_c = '0;

    if(decoded_reg_strb.HPET_COUNTER_LO && decoded_req_is_wr) begin
// SW write
        next_c = (field_storage.HPET_COUNTER_LO.counter_lo.value &
~decoded_wr_biten[31:0]) |
                 (decoded_wr_data[31:0] & decoded_wr_biten[31:0]);
        load_next_c = '1;
    end else begin  // HW write (precedence=sw means HW writes unless
SW writes)
        next_c = hwif_in.HPET_COUNTER_LO.counter_lo.next;
        load_next_c = '1;
    end

    field_combo.HPET_COUNTER_LO.counter_lo.next = next_c;
    field_combo.HPET_COUNTER_LO.counter_lo.load_next = load_next_c;
end

always_ff @(posedge clk) begin
    if(rst) begin
        field_storage.HPET_COUNTER_LO.counter_lo.value <= 32'h0;
    end else begin
        if(field_combo.HPET_COUNTER_LO.counter_lo.load_next) begin
            field_storage.HPET_COUNTER_LO.counter_lo.value <=
field_combo.HPET_COUNTER_LO.counter_lo.next;
        end
    end
end

assign hwif_out.HPET_COUNTER_LO.counter_lo.value =
field_storage.HPET_COUNTER_LO.counter_lo.value;
assign hwif_out.HPET_COUNTER_LO.counter_lo.swmod =
    decoded_reg_strb.HPET_COUNTER_LO && decoded_req_is_wr && |
(decoded_wr_biten[31:0]);
```

1.0.9.3.3 Read Response Logic

PeakRDL generates readback arrays for all registers:

```
// Assign readback values to a flattened array
logic [31:0] readback_array[38];

// Global registers
assign readback_array[0][4:0]   = (decoded_reg_strb.HPET_ID && !
decoded_req_is_wr) ? 5'h0 : '0;
```

```
assign readback_array[0][5:5]   = (decoded_reg_strb.HPET_ID && !
decoded_req_is_wr) ? 1'h1 : '0;
assign readback_array[0][12:8]  = (decoded_reg_strb.HPET_ID && !
decoded_req_is_wr) ?
                                  hwif_in.HPET_ID.num_tim_cap.next :
'0;
assign readback_array[0][23:16] = (decoded_reg_strb.HPET_ID && !
decoded_req_is_wr) ? 8'h1 : '0;
assign readback_array[0][31:24] = (decoded_reg_strb.HPET_ID && !
decoded_req_is_wr) ? 8'h1 : '0;

// Config/status registers
assign readback_array[1][0:0] = (decoded_reg_strb.HPET_CONFIG && !
decoded_req_is_wr) ?

field_storage.HPET_CONFIG.hpet_enable.value : '0;
assign readback_array[2][7:0] = (decoded_reg_strb.HPET_STATUS && !
decoded_req_is_wr) ?

field_storage.HPET_STATUS.timer_int_status.value : '0;

// Counter registers
assign readback_array[4][31:0] = (decoded_reg_strb.HPET_COUNTER_LO
&& !decoded_req_is_wr) ?

field_storage.HPET_COUNTER_LO.counter_lo.value : '0;
assign readback_array[5][31:0] = (decoded_reg_strb.HPET_COUNTER_HI
&& !decoded_req_is_wr) ?

field_storage.HPET_COUNTER_HI.counter_hi.value : '0;

// Per-timer registers
for(genvar i0=0; i0<8; i0++) begin
    assign readback_array[i0 * 4 + 6][2:2] =
(decoded_reg_strb.TIMER[i0].TIMER_CONFIG && !decoded_req_is_wr) ?

field_storage.TIMER[i0].TIMER_CONFIG.timer_enable.value : '0;
    // ... additional timer fields ...
end

// Reduce array via OR (only one element active at a time)
always_comb begin
    automatic logic [31:0] readback_data_var;
    readback_done = decoded_req & ~decoded_req_is_wr;
    readback_err = '0;
    readback_data_var = '0;
    for(int i=0; i<38; i++) readback_data_var |= readback_array[i];
```

```
    readback_data = readback_data_var;
end

assign cpuif_rd_ack = readback_done;
assign cpuif_rd_data = readback_data;
assign cpuif_rd_err = readback_err;
```

1.0.9.4 Field Access Semantics

1.0.9.4.1 Read-Only (RO)

Characteristics: - Software reads return hardware-driven value - Software writes are ignored (no effect) - Hardware controls value via `hwif_in`

Example: HPET_ID register

```
// RO fields: vendor_id, revision_id, num_tim_cap
// Software can read, but writes have no effect
```

1.0.9.4.2 Read-Write (RW)

Characteristics: - Software can read and write - Default next value is current value - Software write updates value - Reset value specified in RDL

Example: HPET_CONFIG.hpet_enable

```
// RW field: Software can enable/disable HPET
// Reset value: 0 (disabled)
```

1.0.9.4.3 Write-1-to-Clear (W1C)

Characteristics: - Software writes 1 to clear bit - Software writes 0 have no effect - Hardware can set bit via `hwif_in.hwset` - Used for sticky interrupt flags

Example: HPET_STATUS.timer_int_status

```
// W1C field: Software writes 1 to clear interrupt
// Hardware sets via hwif_in.HPET_STATUS.timer_int_status.hwset
```

1.0.9.4.4 Hardware Write with Software Precedence

Characteristics: - Hardware continuously writes value via `hwif_in.next` - Software write takes precedence - Used for live counter readback

Example: HPET_COUNTER_LO/HI

```
// hw=w, precedence=sw
// Hardware writes counter value every cycle
// Software write overrides hardware write
```

1.0.9.5 SystemRDL Specification

Source File: `rtl/peakrdl/hpet_regs.rdl`

Key RDL Properties Used:

```
addrmap hpet_regs {
    name = "HPET Register Block";
    desc = "High Precision Event Timer registers";

    default regwidth = 32;       // All registers 32-bit
    default accesswidth = 32;    // Single-beat access

    // Read-only identification
    reg {
        field {
            hw = r;              // Hardware read-only
            sw = r;              // Software read-only
        } vendor_id[31:24] = 8'h01;

        field {
            hw = r; sw = r;
        } revision_id[23:16] = 8'h01;

        field {
            hw = w;              // Hardware controls value
            sw = r;              // Software can only read
        } num_tim_cap[12:8];

    } HPET_ID @ 0x000;

    // Read-write configuration
    reg {
        field {
            sw = rw;             // Software read-write
            hw = r;              // Hardware reads value
        } hpet_enable[0:0] = 1'b0;

        field {
            sw = rw; hw = r;
        } legacy_replacement[1:1] = 1'b0;

    } HPET_CONFIG @ 0x004;

    // Write-1-to-clear status
    reg {
```

```
        field {
            sw = w1c;           // Write 1 to clear
            hw = w;             // Hardware can set
            hwset;              // Hardware set signal available
        } timer_int_status[NUM_TIMERS-1:0];

    } HPET_STATUS @ 0x008;

    // Hardware-written counter with software override
    reg {
        field {
            sw = rw;            // Software can write
            hw = w;             // Hardware writes every cycle
            precedence = sw;    // Software write takes priority
        } counter_lo[31:0] = 32'h0;

    } HPET_COUNTER_LO @ 0x010;

    // Per-timer array
    regfile {
        reg {
            field { sw = rw; hw = r; } timer_enable[2:2] = 1'b0;
            field { sw = rw; hw = r; } timer_int_enable[3:3] = 1'b0;
            field { sw = rw; hw = r; } timer_type[4:4] = 1'b0;
            field { sw = rw; hw = r; } timer_size[5:5] = 1'b0;
            field { sw = rw; hw = r; } timer_value_set[6:6] = 1'b0;
        } TIMER_CONFIG @ 0x00;

        reg {
            field { sw = rw; hw = r; } timer_comp_lo[31:0] = 32'h0;
        } TIMER_COMPARATOR_LO @ 0x04;

        reg {
            field { sw = rw; hw = r; } timer_comp_hi[31:0] = 32'h0;
        } TIMER_COMPARATOR_HI @ 0x08;

    } TIMER[NUM_TIMERS] @ 0x100 += 0x20;  // 32-byte spacing
};
```

1.0.9.6 Regeneration Procedure

When to Regenerate: 1. Changing register addresses 2. Adding/removing fields 3. Modifying field access properties 4. Updating VENDOR_ID, REVISION_ID, or NUM_TIMERS

Steps:

```
cd projects/components/apb_hpet/rtl/peakrdl

# 1. Edit SystemRDL specification
vim hpet_regs.rdl

# 2. Generate RTL
peakrdl regblock hpet_regs.rdl --cpuif passthrough -o ../

# 3. Verify generated files
ls -l ../hpet_regs.sv ../hpet_regs_pkg.sv

# 4. Review changes (if in version control)
git diff ../hpet_regs.sv ../hpet_regs_pkg.sv

# 5. Run tests to verify
pytest projects/components/apb_hpet/dv/tests/ -v
```

⚠️ **Important:** Do not manually edit generated files! All changes must be made in `hpet_regs.rdl` and regenerated.

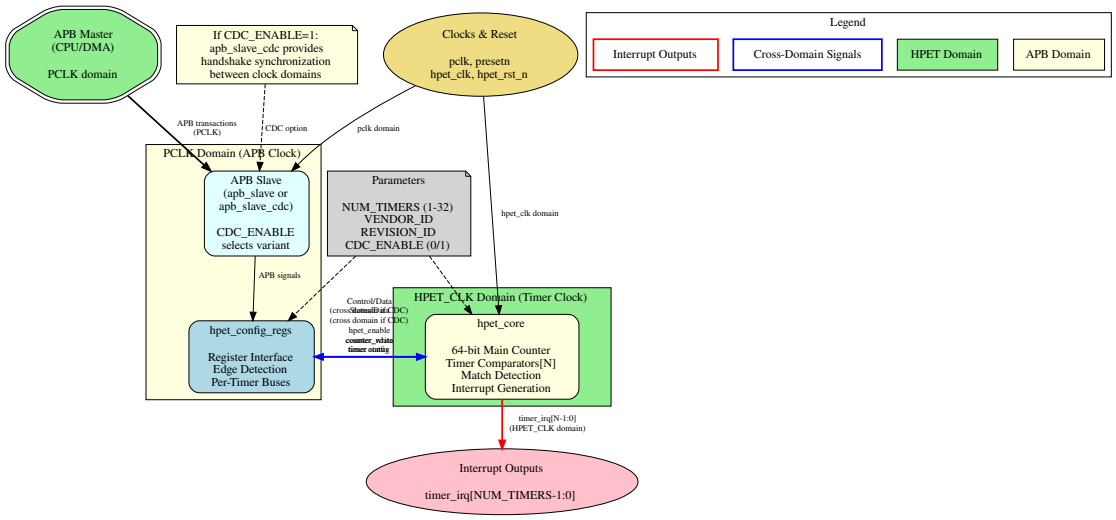
Next: Chapter 2.4 - apb_hpet (Top Level)

1.0.10 APB HPET Top Level - System Integration

1.0.10.1 Overview

The `apb_hpet` module is the top-level system integration point that combines APB slave interface, configuration registers, and timer core into a complete HPET peripheral. It provides parameterized clock domain crossing (CDC) support and exposes a unified external interface.

Top-Level Block Diagram:



APB HPET Top-Level Block Diagram

Figure: APB HPET top-level integration showing dual clock domains (PCLK and HPET_CLK), optional CDC, configuration registers, HPET core, and interrupt outputs. Source: assets/graphviz/apb_hpet.gv | SVG

Key Integration Features: - Conditional APB slave instantiation (CDC or non-CDC) - Clock domain management - Parameter propagation to all child modules - Timer interrupt aggregation - Single-point system configuration

1.0.10.2 Module Hierarchy

```
apb_hpet
+-- APB Slave Interface (conditional generation)
|   +-- apb_slave (CDC_ENABLE=0)
|   |   +-- Synchronous APB protocol
|   +-- apb_slave_cdc (CDC_ENABLE=1)
|       +-- APB protocol (pclk domain)
|       +-- CDC handshake (pclk ↔ hpet_clk)
|
+-- HPET Configuration Registers
|   +-- peakrdl_to_cmdrsp (protocol adapter)
|   +-- hpet_regs (PeakRDL generated)
|   +-- Interface mapping logic
|
+-- HPET Core
    +-- 64-bit counter
    +-- Timer comparators [NUM_TIMERS]
    +-- Interrupt generation [NUM_TIMERS]
```

1.0.10.3 Interface Specification

1.0.10.3.1 Parameters

Parameter	Type	Default	Range	Description
VENDOR_ID	int	1	0-65535	Vendor identification (read-only in HPET_ID register)
REVISION_ID	int	1	0-65535	Revision identification (read-only in HPET_ID register)
NUM_TIMERS	int	2	2, 3, 8	Number of independent timers in array
CDC_ENABLE	int	0	0, 1	Clock domain crossing: 0=synchronous, 1=asynchronous

Parameter Notes: - **VENDOR_ID** and **REVISION_ID**: Informational only, visible in HPET_CAPABILITIES register - **NUM_TIMERS**: Must match PeakRDL generation (currently supports 2, 3, or 8) - **CDC_ENABLE**: Critical for system integration - determines clock relationship

1.0.10.3.2 Clock and Reset - Dual Domain

Signal Name	Type	Width	Direction	Description
pclk	logic	1	Input	APB clock domain (always used for APB interface)
presetn	logic	1	Input	APB reset (active-low)
hpet_clk	logic	1	Input	HPET clock domain (used for timer logic)
hpet_resetn	logic	1	Input	HPET reset (active-low)

Clock Constraints: - **CDC_ENABLE=0:** `pclk` and `hpet_clk` must be the same or synchronous - **CDC_ENABLE=1:** `pclk` and `hpet_clk` can be fully asynchronous

Reset Constraints: - **CDC_ENABLE=0:** `presetn` and `hpet_resetn` should be asserted/deasserted together - **CDC_ENABLE=1:** Both resets must overlap during power-on, can be independent afterward

1.0.10.3.3 APB4 Slave Interface (Low Frequency Domain)

Signal Name	Type	Width	Direction	Description
s_apb_PSEL	logic	1	Input	Peripheral select
s_apb_PENABLE	logic	1	Input	Enable signal
s_apb_PREADY	logic	1	Output	Ready signal
s_apb_PADDR	logic	12	Input	Address bus (fixed 12-bit addressing)
s_apb_PWRITE	logic	1	Input	Write enable (1=write, 0=read)
s_apb_PWDATA	logic	32	Input	Write data bus
s_apb_PSTRB	logic	4	Input	Write strobe (byte enables)
s_apb_PPROT	logic	3	Input	Protection type
s_apb_PRDATA	logic	32	Output	Read data bus
s_apb_PSLVERR	logic	1	Output	Slave error

Address Space: 12-bit addressing supports up to 4KB (0x000-0xFFF) - Global registers: 0x000-0x0FF - Timer registers: 0x100-0x1FF (32-byte spacing per timer) - Reserved: 0x200-0xFFF

1.0.10.3.4 Timer Interrupt Outputs (High Frequency Domain)

Signal Name	Type	Width	Direction	Description
timer_irq[NUM_TIMERS-1:0]	logic	NUM_TIMERS	Output	Per-timer interrupt outputs (active-high)

Interrupt Characteristics: - **Active-high level-sensitive** - **One interrupt per timer** (independent) - **Follows HPET_STATUS register** (sticky until cleared) - **W1C clearing** (software writes 1 to HPET_STATUS to clear)

1.0.10.4 Internal Signal Interfaces

1.0.10.4.1 CDC Command/Response Interface

Between APB Slave and Configuration Registers:

```
logic                   w_cmd_valid;
logic                   w_cmd_ready;
logic                   w_cmd_pwrite;
logic [11:0]            w_cmd_paddr;
logic [31:0]            w_cmd_pwdata;
logic [3:0]             w_cmd_pstrb;
logic [2:0]             w_cmd_pprot;

logic                   w_rsp_valid;
logic                   w_rsp_ready;
logic [31:0]            w_rsp_prdata;
logic                   w_rsp_pslverr;
```

Clock Domain: - **CDC_ENABLE=0:** Runs on `pclk` - **CDC_ENABLE=1:** Runs on `hpet_clk` (synchronized from pclk)

1.0.10.4.2 Configuration Register Interface

Between hpet_config_regs and hpet_core:

```
// Global configuration
logic                   w_hpet_enable;
logic                   w_legacy_replacement;

// Counter interface
logic                   w_counter_write;
logic [63:0]            w_counter_wdata;
logic [63:0]            w_counter_rdata;

// Per-timer configuration
logic [NUM_TIMERS-1:0]  w_timer_enable;
logic [NUM_TIMERS-1:0]  w_timer_int_enable;
logic [NUM_TIMERS-1:0]  w_timer_type;
logic [NUM_TIMERS-1:0]  w_timer_size;
logic [NUM_TIMERS-1:0]  w_timer_value_set;

// Per-timer comparator (dedicated buses)
logic [NUM_TIMERS-1:0]  w_timer_comp_write;
logic [63:0]            w_timer_comp_wdata [NUM_TIMERS];
logic                   w_timer_comp_write_high;
logic [63:0]            w_timer_comp_rdata [NUM_TIMERS];
```

```
// Interrupt status
logic [NUM_TIMERS-1:0]   w_timer_int_status;
logic [NUM_TIMERS-1:0]   w_timer_int_clear;
```

1.0.10.5 APB Slave Conditional Generation

The top-level module uses a SystemVerilog `generate` block to conditionally instantiate the appropriate APB slave variant:

1.0.10.5.1 Non-CDC Configuration (CDC_ENABLE=0)

```
generate
    if (CDC_ENABLE != 0) begin : g_apb_slave_cdc
        // ... CDC variant instantiation ...
    end else begin : g_apb_slave_no_cdc
        // Non-CDC version for same clock domain (pclk == hpet_clk)
        apb_slave #(
            .ADDR_WIDTH(12),
            .DATA_WIDTH(32),
            .STRB_WIDTH(4),
            .PROT_WIDTH(3)
        ) u_apb_slave (
            // Single clock domain (use pclk for both APB and cmd/rsp)
            .pclk                   (pclk),
            .presetn                (presetn),

            // APB Interface
            .s_apb_PSEL             (s_apb_PSEL),
            .s_apb_PENABLE          (s_apb_PENABLE),
            .s_apb_PREADY           (s_apb_PREADY),
            .s_apb_PADDR            (s_apb_PADDR),
            .s_apb_PWRITE           (s_apb_PWRITE),
            .s_apb_PWDATA           (s_apb_PWDATA),
            .s_apb_PSTRB            (s_apb_PSTRB),
            .s_apb_PPROT            (s_apb_PPROT),
            .s_apb_PRDATA           (s_apb_PRDATA),
            .s_apb_PSLVERR          (s_apb_PSLVERR),

            // Command Interface (same pclk domain)
            .cmd_valid              (w_cmd_valid),
            .cmd_ready              (w_cmd_ready),
            .cmd_pwrite             (w_cmd_pwrite),
            .cmd_paddr              (w_cmd_paddr),
            .cmd_pwdata             (w_cmd_pwdata),
            .cmd_pstrb              (w_cmd_pstrb),
            .cmd_pprot              (w_cmd_pprot),

            // Response Interface (same pclk domain)
            .rsp_valid              (w_rsp_valid),
```

```
            .rsp_ready              (w_rsp_ready),
            .rsp_prdata             (w_rsp_prdata),
            .rsp_pslverr            (w_rsp_pslverr)
        );
    end
endgenerate
```

Characteristics: - **Latency:** 2 APB clock cycles (SETUP + ACCESS phases) - **Clock:** Single `pclk` domain - **Resources:** ~20 FF, ~50 LUTs

1.0.10.5.2 CDC Configuration (CDC_ENABLE=1)

```
generate
    if (CDC_ENABLE != 0) begin : g_apb_slave_cdc
        // Clock Domain Crossing version for async clocks
        apb_slave_cdc #(
            .ADDR_WIDTH(12),
            .DATA_WIDTH(32),
            .STRB_WIDTH(4),
            .PROT_WIDTH(3),
            .DEPTH     (2)
        ) u_apb_slave_cdc (
            // APB Clock Domain
            .pclk                   (pclk),
            .presetn                (presetn),

            // HPET Clock Domain
            .aclk                   (hpet_clk),
            .aresetn                (hpet_resetn),

            // APB Interface (pclk domain)
            .s_apb_PSEL             (s_apb_PSEL),
            .s_apb_PENABLE          (s_apb_PENABLE),
            .s_apb_PREADY           (s_apb_PREADY),
            .s_apb_PADDR            (s_apb_PADDR),
            .s_apb_PWRITE           (s_apb_PWRITE),
            .s_apb_PWDATA           (s_apb_PWDATA),
            .s_apb_PSTRB            (s_apb_PSTRB),
            .s_apb_PPROT            (s_apb_PPROT),
            .s_apb_PRDATA           (s_apb_PRDATA),
            .s_apb_PSLVERR          (s_apb_PSLVERR),

            // Command Interface (hpet_clk domain)
            .cmd_valid              (w_cmd_valid),
            .cmd_ready              (w_cmd_ready),
            .cmd_pwrite             (w_cmd_pwrite),
            .cmd_paddr              (w_cmd_paddr),
            .cmd_pwdata             (w_cmd_pwdata),
            .cmd_pstrb              (w_cmd_pstrb),
```

```
            .cmd_pprot          (w_cmd_pprot),

            // Response Interface (hpet_clk domain)
            .rsp_valid          (w_rsp_valid),
            .rsp_ready          (w_rsp_ready),
            .rsp_prdata         (w_rsp_prdata),
            .rsp_pslverr        (w_rsp_pslverr)
        );
    end else begin : g_apb_slave_no_cdc
        // ... non-CDC variant instantiation ...
    end
endgenerate
```

Characteristics: - **Latency:** 4-6 APB clock cycles (CDC handshake overhead) - **Clocks:** Dual domains (pclk and hpet_clk) - **Resources:** ~100 FF, ~150 LUTs (additional CDC logic)

1.0.10.6 Clock Domain Assignment

Configuration registers and HPET core run in a clock domain determined by `CDC_ENABLE`:

```
// HPET Configuration Registers
hpet_config_regs #(
    .VENDOR_ID        (VENDOR_ID),
    .REVISION_ID      (REVISION_ID),
    .NUM_TIMERS       (NUM_TIMERS)
) u_hpet_config_regs (
    // Clock and Reset - conditional based on CDC_ENABLE
    .clk               (CDC_ENABLE[0] ? hpet_clk : pclk),
    .rst_n             (CDC_ENABLE[0] ? hpet_resetn : presetn),
    // ... interface connections ...
);

// HPET Timer Core
hpet_core #(
    .NUM_TIMERS(NUM_TIMERS)
) u_hpet_core (
    // Clock and Reset - conditional based on CDC_ENABLE
    .clk                 (CDC_ENABLE[0] ? hpet_clk : pclk),
    .rst_n               (CDC_ENABLE[0] ? hpet_resetn : presetn),
    // ... interface connections ...
);
```

Clock Assignment Logic: - **CDC_ENABLE=0:** Both use `pclk` and `presetn` - **CDC_ENABLE=1:** Both use `hpet_clk` and `hpet_resetn`

Rationale: Configuration registers and timer core must run in the same domain. APB slave handles the clock crossing (if needed).

1.0.10.7 Integration Examples

1.0.10.7.1 Example 1: Synchronous Configuration (CDC_ENABLE=0)

```
apb_hpet #(
    .VENDOR_ID(16'h8086),      // Intel vendor ID
    .REVISION_ID(16'h0001),
    .NUM_TIMERS(2),
    .CDC_ENABLE(0)              // ← Synchronous clocks
) u_hpet (
    // Use same clock for both domains
    .pclk           (system_clk),
    .presetn        (system_rst_n),
    .hpet_clk       (system_clk),      // ← Same clock as pclk
    .hpet_resetn    (system_rst_n),    // ← Same reset as presetn

    // APB Interface
    .s_apb_PSEL     (apb_psel),
    .s_apb_PENABLE  (apb_penable),
    .s_apb_PREADY   (apb_pready),
    .s_apb_PADDR    (apb_paddr[11:0]),
    .s_apb_PWRITE   (apb_pwrite),
    .s_apb_PWDATA   (apb_pwdata),
    .s_apb_PSTRB    (apb_pstrb),
    .s_apb_PPROT    (apb_pprot),
    .s_apb_PRDATA   (apb_prdata),
    .s_apb_PSLVERR  (apb_pslverr),

    // Timer Interrupts
    .timer_irq      (hpet_irq[1:0])
);

// Connect interrupts to system interrupt controller
assign irq_sources[31:30] = hpet_irq[1:0];
```

1.0.10.7.2 Example 2: Asynchronous Configuration (CDC_ENABLE=1)

```
apb_hpet #(
    .VENDOR_ID(16'h1022),      // AMD vendor ID
    .REVISION_ID(16'h0002),
    .NUM_TIMERS(3),
    .CDC_ENABLE(1)              // ← Asynchronous clocks
) u_hpet (
    // APB domain (slow system clock)
    .pclk           (apb_clk),         // 50 MHz APB clock
    .presetn        (apb_rst_n),

    // HPET domain (high-precision timer clock)
    .hpet_clk       (timer_clk),       // 100 MHz timer clock (async)
```

```
    .hpet_resetn    (timer_rst_n),

    // APB Interface
    .s_apb_PSEL     (apb_psel),
    .s_apb_PENABLE  (apb_penable),
    .s_apb_PREADY   (apb_pready),
    .s_apb_PADDR    (apb_paddr[11:0]),
    .s_apb_PWRITE   (apb_pwrite),
    .s_apb_PWDATA   (apb_pwdata),
    .s_apb_PSTRB    (apb_pstrb),
    .s_apb_PPROT    (apb_pprot),
    .s_apb_PRDATA   (apb_prdata),
    .s_apb_PSLVERR  (apb_pslverr),

    // Timer Interrupts (hpet_clk domain)
    .timer_irq      (hpet_irq[2:0])
);

// Synchronize interrupts to system clock domain
sync_2ff #(.WIDTH(3)) u_irq_sync (
    .i_clk   (system_clk),
    .i_rst_n (system_rst_n),
    .i_data  (hpet_irq[2:0]),
    .o_data  (hpet_irq_sync[2:0])
);

// Connect synchronized interrupts to interrupt controller
assign irq_sources[33:31] = hpet_irq_sync[2:0];
```

1.0.10.8 Resource Utilization Summary

Total Resource Usage by Configuration:

Configuration	NUM_TIMERS	CDC_ENABLE	Flip-Flops	LUTs	BRAM
2-timer sync	2	0	~528 FF	~510 LUTs	0
3-timer sync	3	0	~718 FF	~680 LUTs	0
8-timer sync	8	0	~1544 FF	~1360 LUTs	0
2-timer CDC	2	1	~608 FF	~610 LUTs	0
3-timer	3	1	~798 FF	~780	0

Configuration	NUM_TIMERS	CDC_ENABLE	Flip-Flops	LUTs	BRAM
CDC				LUTs	
8-timer CDC	8	1	~1624 FF	~1460 LUTs	0

Resource Breakdown: - **APB Slave (no CDC):** ~20 FF, ~50 LUTs - **APB Slave CDC:** ~100 FF, ~150 LUTs - **Config Registers:** Scales with NUM_TIMERS (~35 FF + ~70 LUTs per timer) - **HPET Core:** Scales with NUM_TIMERS (~128 FF + ~85 LUTs per timer)

1.0.10.9 Verification Checklist

Integration Validation:

- ☐ **Clock Configuration:**
  - ☐ If CDC_ENABLE=0: Verify pclk = hpet_clk
  - ☐ If CDC_ENABLE=1: Verify independent clock sources
- ☐ **Reset Coordination:**
  - ☐ Both resets overlap at power-on
  - ☐ Both resets held for >=10 cycles
  - ☐ Reset deasserted cleanly
- ☐ **APB Interface:**
  - ☐ Read/write to all registers functional
  - ☐ Address decoding correct
  - ☐ PREADY timing appropriate (2 cycles sync, 4-6 cycles CDC)
- ☐ **Timer Operation:**
  - ☐ All NUM_TIMERS functional
  - ☐ One-shot mode works
  - ☐ Periodic mode works
  - ☐ Counter increments correctly
- ☐ **Interrupt Generation:**
  - ☐ All timer_irq outputs functional
  - ☐ W1C clearing works
  - ☐ Sticky behavior correct
- ☐ **CDC (if enabled):**
  - ☐ No metastability issues
  - ☐ Data integrity across domains

- ☐ Proper handshake protocol

Next: Chapter 2.5 - FSM Summary

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

1.0.11 APB HPET - FSM Summary

1.0.11.1 Finite State Machines Overview

The APB HPET component contains multiple state machines across different modules. This chapter summarizes all FSMs, their states, transitions, and interactions.

1.0.11.2 FSM Inventory

Module	FSM Name	Type	States	Purpose
apb_slave	APB Protocol FSM	Explicit	2-3	APB handshake protocol
apb_slave_cdc	CDC Handshake FSM	Explicit	4	Clock domain crossing protocol
hpet_core	Per-Timer FSM	Conceptual	5	Timer operation and fire control

Note: The hpet_config_regs and hpet_regs modules use combinational and sequential logic without explicit state machines.

1.0.12 1. APB Slave Protocol FSM

Module: `apb_slave.sv` **Clock Domain:** `pclk` **Implementation:** Explicit state register

1.0.12.1 States

State	Encoding	Description
IDLE	2'b00	Waiting for PSEL assertion

State	Encoding	Description
SETUP	2'b01	PSEL asserted, waiting for PENABLE
ACCESS	2'b10	PENABLE asserted, transaction active

1.0.12.2 State Transitions

IDLE -> SETUP: - **Condition:** `PSEL = 1` - **Action:** Latch address, write data, and control signals - **Duration:** 1 clock cycle

SETUP -> ACCESS: - **Condition:** `PENABLE = 1` (always follows SETUP in next cycle) - **Action:** Assert cmd_valid to downstream, wait for rsp_valid - **Duration:** Variable (1 cycle minimum, waits for rsp_valid)

ACCESS -> IDLE: - **Condition:** `rsp_valid = 1` (response received) - **Action:** Assert PREADY, complete transaction - **Duration:** Immediate return to IDLE

ACCESS -> IDLE (Early Termination): - **Condition:** `PSEL = 0` (transaction aborted) - **Action:** Deassert cmd_valid, return to IDLE - **Duration:** Immediate

1.0.12.3 Timing Diagram

```
Clock:      -+ +-+ +-+ +-+ +-+ +-
pclk        +-+ +-+ +-+ +-+ +-

PSEL:       ---+       +-------
            +-----------+

PENABLE:    -------+   +-------
            +-----------+

State:      [IDLE][SETUP][ACCESS][IDLE]

PREADY:     -----------+ +-----
            +-----------+

Latency: 2 cycles (SETUP + ACCESS)
```

1.0.13 2. APB Slave CDC Handshake FSM

Module: `apb_slave_cdc.sv` **Clock Domains:** `pclk` (APB side) and `aclk` (application side) **Implementation:** Dual FSMs with handshake synchronization

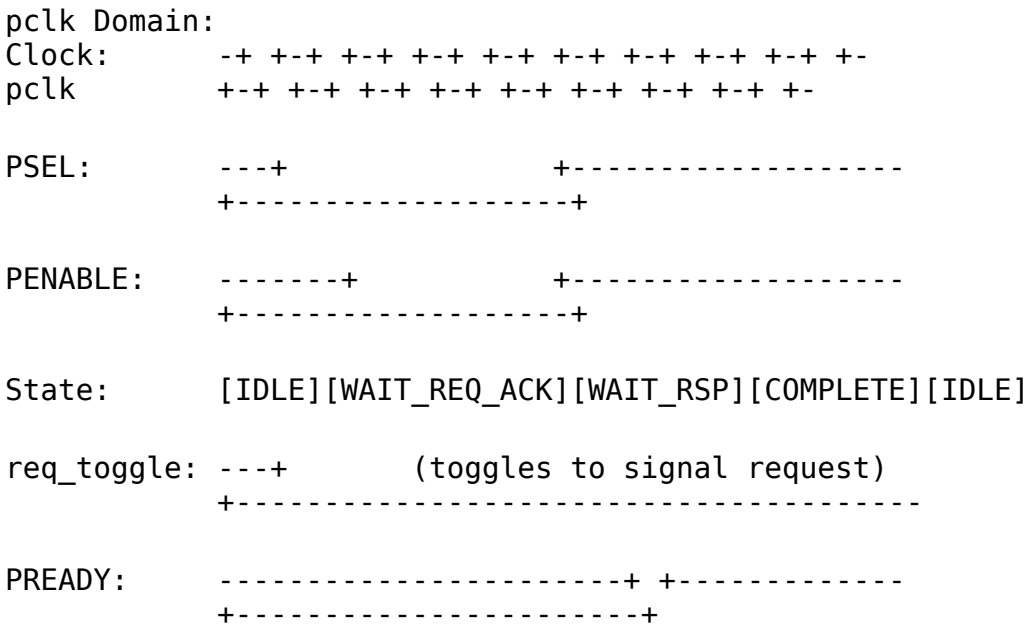
1.0.13.1 pclk Domain States

State	Encoding	Description
IDLE	2'b00	Waiting for APB transaction
WAIT_REQ_ACK	2'b01	Request sent, waiting for ACK from aclk domain
WAIT_RSP	2'b10	ACK received, waiting for response from aclk domain
COMPLETE	2'b11	Response received, completing APB transaction

1.0.13.2 aclk Domain States

State	Encoding	Description
IDLE	2'b00	Waiting for synchronized request from pclk domain
REQ_PEND	2'b01	Request detected, processing command
WAIT_APP_RSP	2'b10	Command sent to application, waiting for response
RSP_READY	2'b11	Response ready, waiting for pclk domain acknowledgment

1.0.13.3 Cross-Domain Handshake Timing



```
aclk Domain:
Clock:      --+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-
aclk        +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-

State:      [IDLE][REQ_PEND][WAIT_APP_RSP][RSP_READY][IDLE]

cmd_valid:  -------+       +-----------------------
            +---------------+

rsp_valid:  ---------------+       +---------------
            +-----------------------+

ack_toggle: -------------------+   (toggles to ack response)
            +--------------------------------------

Latency: 4-6 pclk cycles (depending on clock ratios)
```

Key Mechanisms: - **Toggle-based handshake:** Avoids pulse synchronization issues - **2-stage synchronizers:** All cross-domain signals synchronized - **Request/acknowledge protocol:** Ensures data stability before sampling

---

1.0.143. HPET Core Per-Timer FSM

Module: `hpet_core.sv` **Clock Domain:** `hpet_clk` (or `pclk` if CDC_ENABLE=0) **Implementation:** Conceptual FSM (implemented as combinational logic, not explicit state register)

Note: The HPET core uses a conceptual FSM model for specification clarity, but the actual implementation uses combinational logic and edge detection rather than explicit state registers. This provides simpler timing and resource usage while maintaining the same functional behavior.

1.0.14.1 States

State	Description	Duration
IDLE	Timer disabled, waiting for enable signal	Until timer enabled
ARMED	Timer enabled, monitoring counter vs comparator	Until counter match
FIRE	Timer match detected, asserting interrupt	1 cycle (edge-detected)
PERIODIC_RELOAD	Periodic mode: auto-increment comparator	1 cycle

State	Description	Duration
ONE_SHOT_COMPLETE	One-shot mode: timer complete, waiting for reconfigure	Until STATUS cleared or timer disabled

1.0.14.2 State Transition Conditions

IDLE -> ARMED: - **Condition:** `hpet_enable = 1 AND timer_enable[i] = 1` - **Action:** Latch current comparator value, begin monitoring - **Trigger:** Rising edge of enable signals

ARMED -> FIRE: - **Condition:** `counter >= comparator[i]` - **Action:** Assert `timer_fired[i]` flag, generate interrupt - **Trigger:** Counter comparison (combinational)

FIRE -> PERIODIC_RELOAD: - **Condition:** `timer_type[i] = 1` (periodic mode) - **Action:** `comparator[i] <= comparator[i] + period[i]` - **Trigger:** Immediate (next clock cycle after fire)

FIRE -> ONE_SHOT_COMPLETE: - **Condition:** `timer_type[i] = 0` (one-shot mode) - **Action:** Hold `timer_fired[i]` flag, interrupt remains asserted - **Trigger:** Immediate (next clock cycle after fire)

PERIODIC_RELOAD -> ARMED: - **Condition:** Always (automatic transition) - **Action:** Resume monitoring with new comparator value - **Trigger:** Immediate (next clock cycle)

ONE_SHOT_COMPLETE -> ARMED: - **Condition:** `timer_comparator_wr[i] = 1` (software reconfigures comparator) - **Action:** Resume monitoring with new comparator value - **Trigger:** Comparator write strobe

ARMED -> IDLE: - **Condition:** `hpet_enable = 0 OR timer_enable[i] = 0` - **Action:** Clear timer state, stop monitoring - **Trigger:** Falling edge of enable signals

ONE_SHOT_COMPLETE -> IDLE: - **Condition:** `timer_enable[i] = 0` - **Action:** Clear timer state - **Trigger:** Timer disable

1.0.14.3 FSM Timing Examples

One-Shot Mode:

```
Clock:      -+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-
hpet_clk    +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-

Enable:     ---+                        +-------------------
timer_enable   +------------------------+

Counter:    [0] [1] [2] [3] [4] [5] [6] [0] [1] [2] [3]

Comparator: [5] [5] [5] [5] [5] [5] [5] [5] [5] [5] [5]

State:      [IDLE][ARMED][ARMED][ARMED][ARMED][FIRE]
```

```
[ONE_SHOT_COMPLETE][IDLE]

timer_fired:---------------------+           +-------
            +-----------------------------------+

timer_irq:  ---------------------+           +-------
            +-----------------------------------+

Status Clear:------------------------------+ +-
            +------------------------------+

Note: Fire at counter=5, interrupt sticky until status cleared
```

Periodic Mode:

```
Clock:      -+ +-+ +-+ + +-+ +-+ + +-+ +-+ + +-+ +-
hpet_clk    +-+ +-+ +- +-+ +-+ +- +-+ +-+ +- +-+ +-

Counter:    [8] [9] [10][11][12][13][14][15][16][17]

Comparator: [10][10][10][13][13][13][16][16][16][19]
                    ↑       ↑       ↑
                Fire 1  Fire 2  Fire 3

State:      [ARMED][ARMED][FIRE][RELOAD][ARMED][ARMED][FIRE]
[RELOAD]...

timer_fired:--------+ +---------+ +---------+ +---
            +-------+ +---------+ +---------+

timer_irq:  --------+ +---------+ +---------+ +---
            +-------+ +---------+ +---------+

Period:     [3] [3] [3] [3] [3] [3] [3] [3] [3] [3]

Note: Fire every 3 counts, comparator auto-increments by period
```

1.0.15 FSM Interaction Summary

1.0.15.1 Cross-Module State Dependencies

```
APB Transaction Flow:
APB Slave FSM (pclk)
    ↓ cmd_valid
hpet_config_regs (combinational mapping)
```

```
    ↓ timer_enable, timer_comparator_wr
HPET Core Timer FSM (hpet_clk)
    ↓ timer_fired
hpet_config_regs (interrupt edge detection)
    ↓ hwif_in.timer_int_status.hwset
PeakRDL Registers (status latch)
    ← software read HPET_STATUS
    ← software write W1C to clear
    ↓ hwif_out.timer_int_status.swmod
hpet_config_regs (clear pulse generation)
    ↓ timer_int_clear
HPET Core Timer FSM
    -> timer_fired clears
```

1.0.15.2 Clock Domain Considerations

Synchronous Mode (CDC_ENABLE=0): - All FSMs run on `pclk` - No synchronization required - Direct signal propagation

Asynchronous Mode (CDC_ENABLE=1): - APB Slave CDC FSM bridges `pclk` and `hpet_clk` - Configuration registers and timers run on `hpet_clk` - Handshake protocol ensures data stability

---

1.0.16 State Machine Design Patterns

1.0.16.1 Pattern 1: Explicit State Register (APB Slave)

```
typedef enum logic [1:0] {
    IDLE   = 2'b00,
    SETUP  = 2'b01,
    ACCESS = 2'b10
} state_t;

state_t r_state, w_next_state;

always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn) r_state <= IDLE;
    else          r_state <= w_next_state;
end

always_comb begin
    w_next_state = r_state;  // Default: hold state
    case (r_state)
        IDLE:   if (PSEL)              w_next_state = SETUP;
        SETUP:  if (PENABLE)           w_next_state = ACCESS;
        ACCESS: if (rsp_valid || !PSEL) w_next_state = IDLE;
    endcase
end
```

Characteristics: - Explicit state storage - Separate combo/sequential blocks - Easy to verify and debug - Standard FSM coding style

1.0.16.2 Pattern 2: Combinational Logic with Edge Detection (Timer FSM)

```
// No explicit state register - use combinational logic + edge detect

// Current match condition
assign w_timer_match[i] = (counter >= comparator[i]) &&
timer_enable[i] && hpet_enable;

// Previous match state (for edge detection)
always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n) r_timer_match_prev[i] <= 1'b0;
    else             r_timer_match_prev[i] <= w_timer_match[i];
end

// Fire edge (rising edge of match)
assign w_timer_fire_edge[i] = w_timer_match[i] && !
r_timer_match_prev[i];

// Fire flag storage (sticky vs pulse based on mode)
always_ff @(posedge hpet_clk or negedge hpet_rst_n) begin
    if (!hpet_rst_n || !timer_enable[i]) begin
        r_timer_fired[i] <= 1'b0;
    end else if (w_timer_fire_edge[i]) begin
        r_timer_fired[i] <= 1'b1;
    end else if (timer_type[i]) begin  // Periodic: clear after 1
cycle
        r_timer_fired[i] <= 1'b0;
    end
    // One-shot: hold until status cleared (implicit)
end
```

Characteristics: - No explicit state register - Edge detection for transitions - Simpler implementation - Lower resource usage - Same functional behavior as FSM

1.0.17 FSM Verification Considerations

1.0.17.1 State Coverage

APB Slave FSM: - [ ] IDLE state entry and exit - [ ] SETUP state timing (1 cycle) - [ ] ACCESS state with response wait - [ ] ACCESS state early termination (PSEL deassert)

CDC Handshake FSM: - [ ] Request synchronization (pclk -> aclk) - [ ] Response synchronization (aclk -> pclk) - [ ] Concurrent requests handling - [ ] Clock ratio corner cases (fast pclk, slow aclk and vice versa)

Timer FSM: - [ ] IDLE -> ARMED transition - [ ] ARMED -> FIRE on match - [ ] FIRE -> PERIODIC_RELOAD path - [ ] FIRE -> ONE_SHOT_COMPLETE path - [ ] PERIODIC_RELOAD -> ARMED auto-transition - [ ] ONE_SHOT_COMPLETE -> ARMED on reconfigure - [ ] Return to IDLE on disable

1.0.17.2 Transition Coverage

Edge Cases: - [ ] Enable/disable during active timer - [ ] Comparator write during countdown - [ ] Counter write during active timer - [ ] Multiple timers firing simultaneously - [ ] Interrupt clear during fire event - [ ] Mode switch (one-shot ↔️ periodic) mid-operation

---

Next: Chapter 3 - Interfaces

RTL Design Sherpa · Learning Hardware Design Through Practice GitHub · Documentation Index · MIT License

---

2 APB HPET Register Map

Chapter: 5.1 **Title:** Complete Register Address Map **Version:** 1.0 **Last Updated:** 2025-10-20

2.1 Overview

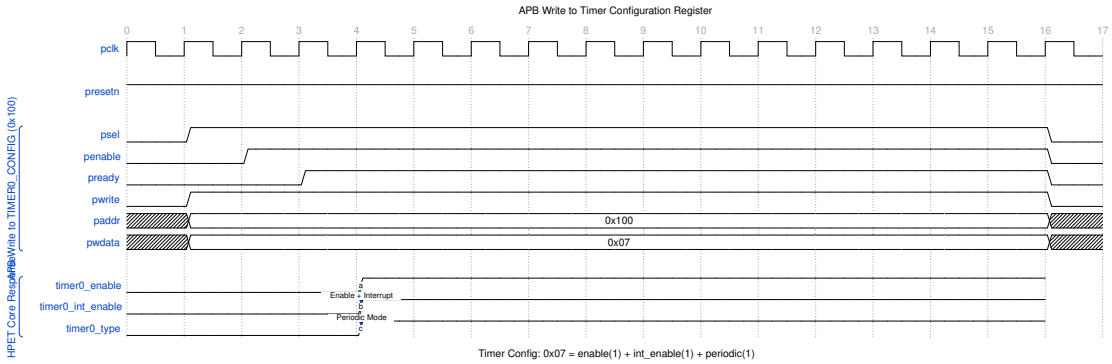
The APB HPET provides a memory-mapped register interface accessible via the APB slave port. The register space is organized into two main sections:

1. **Global Registers (0x000-0x0FF):** Configuration, status, and main counter
2. **Per-Timer Registers (0x100-0x1FF):** Timer-specific configuration and comparators

Each timer occupies a 32-byte (0x20) register block, supporting up to 8 timers.

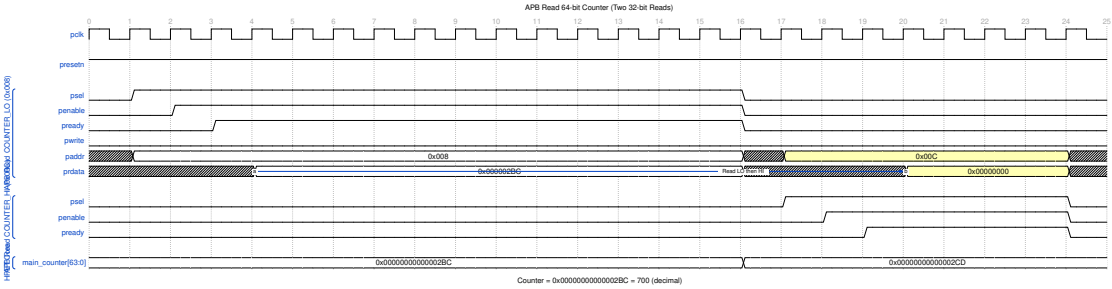
Timing Diagrams:

The following timing diagrams illustrate key register access sequences:



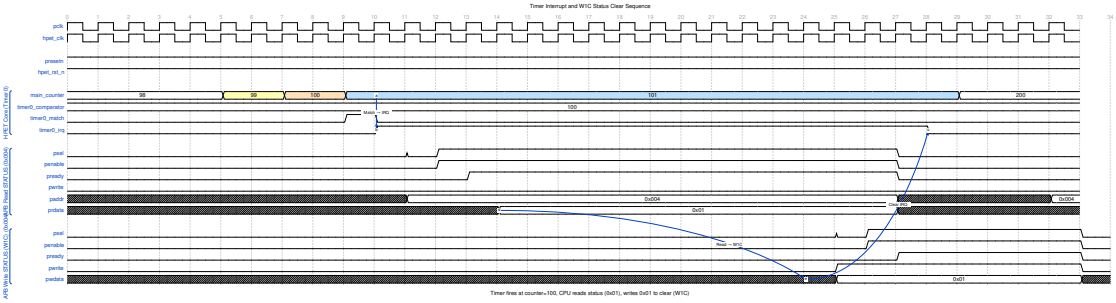
APB Write Timer Config

Figure 1: APB write to TIMER0_CONFIG register (0x100). *Source: assets/wavedrom/apb_write_timer_config.json*



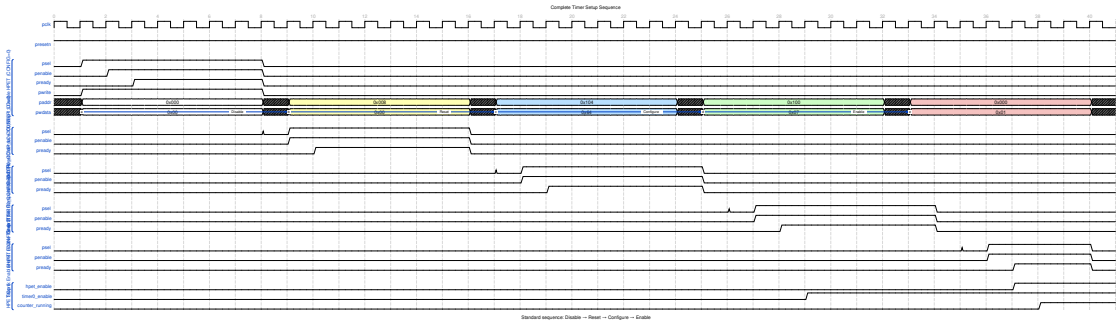
APB Read Counter

Figure 2: APB read of 64-bit counter (two 32-bit reads from COUNTER_LO and COUNTER_HI). *Source: assets/wavedrom/apb_read_counter.json*



Interrupt W1C Sequence

Figure 3: Timer interrupt generation and W1C (Write-1-to-Clear) status clearing sequence. *Source: assets/wavedrom/interrupt_w1c_sequence.json*



Timer Setup Sequence

Figure 4: Complete timer setup sequence: disable HPET, reset counter, configure comparator, enable timer, enable HPET. Source: assets/wavedrom/timer_setup_sequence.json

2.1.1 Block Diagram

APB HPET Block Diagram

APB HPET Block Diagram

Figure 1: APB HPET top-level architecture showing APB interface, configuration registers, HPET core, and timer outputs.

2.2 Register Address Map Summary

2.2.1 Global Registers

Offset	Register Name	Access	Width	Description
0x000	HPET_ID	RO	32b	Identification register (vendor, revision, capabilities)
0x004	HPET_CONFIG	RW	32b	Global configuration and control
0x008	HPET_STATUS	RW/W1C	32b	Interrupt status for all timers (write-1-to-clear)
0x00C	RESERVED	RO	32b	Reserved
0x010	HPET_COUNTER_LO	RW	32b	Main counter bits

Offset	Register Name	Access	Width	Description
				[31:0]
0x014	HPET_COUNTER_HI	RW	32b	Main counter bits [63:32]
0x018-0x0FF	RESERVED	RO	-	Reserved for future use

2.2.2 Per-Timer Registers

Each timer (N = 0 to NUM_TIMERS-1) has a 32-byte register block at base address `0x100 + N*0x20`.

Timer N Base Address: `0x100 + N * 0x20`

Offset	Register Name	Access	Width	Description
+0x00	TIMER_CONFIG	RW	32b	Timer configuration and control
+0x04	TIMER_COMPARATOR_LO	RW	32b	Timer comparator bits [31:0]
+0x08	TIMER_COMPARATOR_HI	RW	32b	Timer comparator bits [63:32]
+0x0C	RESERVED	RO	32b	Reserved
+0x10-0x1F	RESERVED	RO	-	Reserved for timer expansion

Example Timer Addresses:

Timer	Base Address	CONFIG	COMPARATOR_LO	COMPARATOR_HI
0	0x100	0x100	0x104	0x108
1	0x120	0x120	0x124	0x128
2	0x140	0x140	0x144	0x148
3	0x160	0x160	0x164	0x168
4	0x180	0x180	0x184	0x188
5	0x1A0	0x1A0	0x1A4	0x1A8
6	0x1C0	0x1C0	0x1C4	0x1C8
7	0x1E0	0x1E0	0x1E4	0x1E8

Timer	Base Address	CONFIG	COMPARATOR_LO	COMPARATOR_HI

2.3 Global Register Descriptions

2.3.1 HPET_ID (0x000) - Identification Register

Access: Read-Only **Reset Value:** Parameterized (VENDOR_ID, REVISION_ID, NUM_TIMERS)

Contains capability information and identification fields.

Bits	Field	Access	Reset	Description
[31:24]	vendor_id	RO	VENDOR_ID	Vendor identifier (parameterized)
[23:16]	rev_id	RO	REVISION_ID	Revision identifier (parameterized)
[15:13]	reserved	RO	0	Reserved
[12:8]	num_tim_cap	RO	NUM_TIMERS-1	Number of timers minus 1 (e.g., 7 for 8 timers)
[7]	count_size_cap	RO	1	Counter size capability (1 = 64-bit counter)
[6]	reserved	RO	0	Reserved
[5]	leg_rt_cap	RO	1	Legacy replacement capable (1 = supported)
[4:0]	reserved	RO	0	Reserved

Example Values: - 2-timer Intel-like: `0x80860001_00000171` (vendor=0x8086, rev=1, timers=1) - 3-timer AMD-like: `0x10220002_00000271` (vendor=0x1022, rev=2, timers=2) - 8-timer custom: `0x12340001_000007F1` (vendor=0x1234, rev=1, timers=7)

2.3.2 HPET_CONFIG (0x004) - Configuration Register

Access: Read-Write **Reset Value:** 0x00000000

Global enable and configuration control.

Bits	Field	Access	Reset	Description
[31:2]	reserved	RO	0	Reserved
[1]	legacy_replacement	RW	0	Legacy replacement mode enable (0=disabled, 1=enabled)
[0]	hpet_enable	RW	0	HPET main counter enable (0=stopped, 1=running)

Usage Notes: - Write `hpet_enable=1` to start the main counter - Write `hpet_enable=0` to stop the main counter (value preserved) - `legacy_replacement` enables mapping to legacy timer interrupt lines (implementation-specific) - Counter must be enabled for any timer to fire

Example Configuration Sequence:

```
// Disable HPET
WRITE(HPET_CONFIG, 0x0);

// Reset counter
WRITE(HPET_COUNTER_LO, 0x0);
WRITE(HPET_COUNTER_HI, 0x0);

// Configure timers...

// Enable HPET
WRITE(HPET_CONFIG, 0x1);
```

2.3.3 HPET_STATUS (0x008) - Interrupt Status Register

Access: Read-Write (Write-1-to-Clear) **Reset Value:** 0x00000000

Interrupt status bits for all timers. Write 1 to a bit to clear the corresponding interrupt.

Bits	Field	Access	Reset	Description
[31:NUM_TIMERS]	reserved	RO	0	Reserved (unused timer bits)
[NUM_TIMERS-1:0]	timer_int_status	RW/W1C	0	Timer interrupt status bits

Per-Timer Status Bit: - **Bit[N]** = Timer N interrupt status - 0 = No interrupt pending - 1 = Timer N has fired, interrupt pending

Write-1-to-Clear (W1C) Behavior: - Write 1 to bit[N] to clear Timer N interrupt status - Write 0 has no effect - Reading returns current interrupt status

Example Interrupt Handling:

```
// Read interrupt status
uint32_t status = READ(HPET_STATUS);

// Check if Timer 0 fired
if (status & 0x1) {
    // Handle Timer 0 interrupt

    // Clear Timer 0 interrupt
    WRITE(HPET_STATUS, 0x1);  // Write 1 to clear bit 0
}

// Clear all pending interrupts
WRITE(HPET_STATUS, status);  // Write back read value clears all set
bits
```

2.3.4 HPET_COUNTER_LO (0x010) - Main Counter Low

Access: Read-Write **Reset Value:** 0x00000000

Lower 32 bits of the 64-bit free-running main counter.

Bits	Field	Access	Reset	Description
[31:0]	counter_lo	RW	0	Main counter bits [31:0]

Behavior: - **Read:** Returns current counter value [31:0] - **Write:** Sets counter value [31:0] (writes both LO and HI together) - Counter increments every `hpet_clk` cycle when `HPET_CONFIG.hpet_enable=1` - Software can write to reset or set counter to specific value

Usage Notes: - Writing counter is useful for test/debug or implementing periodic reset - When writing 64-bit counter, write LO first, then HI - Counter write takes effect immediately (on next `hpet_clk`) - All timers compare against this counter value

---

2.3.5 HPET_COUNTER_HI (0x014) - Main Counter High

Access: Read-Write **Reset Value:** 0x00000000

Upper 32 bits of the 64-bit free-running main counter.

Bits	Field	Access	Reset	Description
[31:0]	counter_hi	RW	0	Main counter bits [63:32]

Behavior: - Same as HPET_COUNTER_LO but for upper 32 bits - Forms complete 64-bit counter value: `{counter_hi, counter_lo}`

Reading 64-bit Counter:

```
// Read lower 32 bits first (in case of rollover during read)
uint32_t lo = READ(HPET_COUNTER_LO);
uint32_t hi = READ(HPET_COUNTER_HI);
uint64_t counter = ((uint64_t)hi << 32) | lo;
```

Writing 64-bit Counter:

```
// Write lower 32 bits first, then upper
WRITE(HPET_COUNTER_LO, 0x00000000);
WRITE(HPET_COUNTER_HI, 0x00000000);
```

2.4 Per-Timer Register Descriptions

Each timer has a dedicated 32-byte register block. The following descriptions apply to Timer N at base address `0x100 + N*0x20`.

2.4.1 TIMER_CONFIG (Timer Base + 0x00) - Timer Configuration

Access: Read-Write **Reset Value:** 0x00000000

Configuration and control for individual timer.

Bits	Field	Access	Reset	Description
[31:7]	reserved	RO	0	Reserved
[6]	timer_value_set	RW	0	Write 1 to set timer value (implementation-specific)
[5]	timer_size	RW	0	Timer size (0=32-bit,

Bits	Field	Access	Reset	Description
				1=64-bit)
[4]	timer_type	RW	0	Timer mode (0=one-shot, 1=periodic)
[3]	timer_int_enable	RW	0	Interrupt enable (0=disabled, 1=enabled)
[2]	timer_enable	RW	0	Timer enable (0=disabled, 1=enabled)
[1:0]	reserved	RO	0	Reserved

Field Descriptions:

timer_enable (bit 2): - 0 = Timer disabled (comparator inactive) - 1 = Timer enabled (comparator active) - Timer only fires when enabled AND `HPET_CONFIG.hpet_enable=1`

timer_int_enable (bit 3): - 0 = Interrupt generation disabled (timer fires but no interrupt) - 1 = Interrupt generation enabled (sets `HPET_STATUS` bit on fire)

timer_type (bit 4): - 0 = **One-shot mode:** Timer fires once when counter >= comparator, then stays idle - 1 = **Periodic mode:** Timer fires repeatedly, auto-increments comparator by period

timer_size (bit 5): - 0 = 32-bit timer (uses only COMPARATOR_LO, ignores COMPARATOR_HI) - 1 = 64-bit timer (uses full 64-bit comparator) - APB HPET supports 64-bit by default

timer_value_set (bit 6): - Implementation-specific flag for timer value updates - Writing 1 may trigger immediate comparator reload (implementation-dependent)

Common Configurations:

```
// One-shot timer with interrupt
WRITE(TIMER0_CONFIG, 0x0C);  // bits [3:2] = enable | int_enable

// Periodic timer with interrupt
WRITE(TIMER0_CONFIG, 0x1C);  // bits [4:3:2] = periodic | int_enable |
enable

// One-shot timer, 64-bit, with interrupt
WRITE(TIMER0_CONFIG, 0x2C);  // bits [5:3:2] = 64-bit | int_enable |
enable
```

2.4.2 TIMER_COMPARATOR_LO (Timer Base + 0x04) - Comparator Low

Access: Read-Write **Reset Value:** 0x00000000

Lower 32 bits of the 64-bit timer comparator value.

Bits	Field	Access	Reset	Description
[31:0]	timer_comp_lo	RW	0	Timer comparator bits [31:0]

Behavior: - Timer fires when `main_counter >= comparator` - For **one-shot mode:** Comparator value stays unchanged after fire - For **periodic mode:** Comparator auto-increments by period value on fire - Software writes to set initial comparator value

Usage:

```
// Set Timer 0 to fire at 1000 cycles (assuming HPET_clk = counter increment)
WRITE(TIMER0_COMPARATOR_LO, 1000);
WRITE(TIMER0_COMPARATOR_HI, 0);
```

2.4.3 TIMER_COMPARATOR_HI (Timer Base + 0x08) - Comparator High

Access: Read-Write **Reset Value:** 0x00000000

Upper 32 bits of the 64-bit timer comparator value.

Bits	Field	Access	Reset	Description
[31:0]	timer_comp_hi	RW	0	Timer comparator bits [63:32]

Behavior: - Forms complete 64-bit comparator: `{timer_comp_hi, timer_comp_lo}` - Same behavior as COMPARATOR_LO but for upper 32 bits

64-bit Timer Example:

```
// Set Timer 1 to fire at 0x0000_0001_0000_0000 (4.3 billion cycles)
WRITE(TIMER1_COMPARATOR_LO, 0x00000000);
WRITE(TIMER1_COMPARATOR_HI, 0x00000001);
```

2.5 Timer Operation Modes

2.5.1 One-Shot Mode (timer_type = 0)

One-Shot Timer Operation

One-Shot Timer Operation

Figure 2: One-shot timer operation flow showing counter increment, comparator match, and idle state after fire.

Behavior: 1. Counter increments: 0 → 1 → 2 → ... → comparator 2. When `counter >= comparator`: Timer fires (edge detection 0→1) 3. If `timer_int_enable=1`: Sets `HPET_STATUS` bit 4. Timer stays idle (must reconfigure to fire again)

Comparator Behavior: - Stays unchanged after fire - Software must write new comparator value to re-arm timer

Use Cases: - Single timeout events - Software-initiated timing - Watchdog timers (with software reload)

Example:

```
// Configure Timer 0: One-shot, 1000 cycles
WRITE(TIMER0_COMPARATOR_LO, 1000);
WRITE(TIMER0_CONFIG, 0x0C);  // enable | int_enable

// Enable HPET
WRITE(HPET_CONFIG, 0x1);

// Wait for interrupt
while (!(READ(HPET_STATUS) & 0x1));

// Clear interrupt
WRITE(HPET_STATUS, 0x1);

// Re-arm for next fire at 2000 cycles
WRITE(TIMER0_COMPARATOR_LO, 2000);
WRITE(TIMER0_CONFIG, 0x0C);
```

2.5.2 Periodic Mode (timer_type = 1)

Periodic Timer Operation

Periodic Timer Operation

Figure 3: Periodic timer operation flow showing counter increment, comparator match, auto-increment, and continuous firing.

Behavior: 1. Counter increments: 0 → 1 → 2 → ... → comparator 2. When `counter >= comparator`: Timer fires (edge detection 0→1) 3. If `timer_int_enable=1`: Sets `HPET_STATUS` bit 4. **Comparator auto-increments:** `comparator = comparator + period` 5. Timer repeats indefinitely (fires at 1×period, 2×period, 3×period, ...)

Comparator Auto-Increment: - Hardware automatically adds period value to comparator - Period = initial comparator value written by software - Example: Initial comparator = 1000 → Fires at 1000, 2000, 3000, ...

Use Cases: - Periodic interrupts (e.g., 1 kHz tick) - PWM generation - Periodic data sampling - Heartbeat signals

Example:

```
// Configure Timer 1: Periodic, 2000 cycle period
WRITE(TIMER1_COMPARATOR_LO, 2000);  // Initial comparator = period
WRITE(TIMER1_CONFIG, 0x1C);  // periodic | int_enable | enable

// Enable HPET
WRITE(HPET_CONFIG, 0x1);

// Timer fires at:
// - 2000 cycles (counter >= 2000)
// - 4000 cycles (counter >= 4000) [comparator auto-incremented to
4000]
// - 6000 cycles (counter >= 6000) [comparator auto-incremented to
6000]
// - ... indefinitely

// Interrupt handler
void timer1_isr(void) {
    // Clear interrupt
    WRITE(HPET_STATUS, 0x2);  // Clear bit 1 (Timer 1)

    // Handle periodic event
    // ...

    // No need to reconfigure - timer continues automatically
}
```

2.6 Register Access Examples

2.6.1 Initialization Sequence

Software Initialization Flow

Software Initialization Flow

Figure 4: Software initialization sequence showing configuration steps from disable to enable.

```
// 1. Disable HPET
WRITE(HPET_CONFIG, 0x0);

// 2. Reset main counter
WRITE(HPET_COUNTER_LO, 0x0);
WRITE(HPET_COUNTER_HI, 0x0);

// 3. Configure Timer 0 (one-shot, 10ms @ 10MHz)
WRITE(TIMER0_COMPARATOR_LO, 100000);  // 100,000 cycles = 10ms
WRITE(TIMER0_COMPARATOR_HI, 0x0);
WRITE(TIMER0_CONFIG, 0x0C);  // enable | int_enable

// 4. Configure Timer 1 (periodic, 1ms @ 10MHz)
WRITE(TIMER1_COMPARATOR_LO, 10000);  // 10,000 cycles = 1ms period
WRITE(TIMER1_COMPARATOR_HI, 0x0);
WRITE(TIMER1_CONFIG, 0x1C);  // periodic | int_enable | enable

// 5. Enable HPET
WRITE(HPET_CONFIG, 0x1);
```

2.6.2 Reading Capabilities

```
// Read identification register
uint32_t id = READ(HPET_ID);

// Extract fields
uint8_t vendor_id = (id >> 24) & 0xFF;
uint8_t rev_id = (id >> 16) & 0xFF;
uint8_t num_timers = ((id >> 8) & 0x1F) + 1;  // num_tim_cap + 1
uint8_t is_64bit = (id >> 7) & 0x1;
uint8_t leg_cap = (id >> 5) & 0x1;

printf("HPET: Vendor=0x%02X, Rev=%d, Timers=%d, 64-bit=%d\n",
       vendor_id, rev_id, num_timers, is_64bit);
```

2.6.3 Interrupt Handling

Interrupt Handling Flow

Interrupt Handling Flow

Figure 5: Interrupt handling flow showing status check, handler dispatch, and W1C clear sequence.

```
// Generic interrupt handler
void hpet_interrupt_handler(void) {
    // Read status register
    uint32_t status = READ(HPET_STATUS);

    // Check which timers fired
```

```
    if (status & (1 << 0)) {
        // Timer 0 fired
        handle_timer0();
        WRITE(HPET_STATUS, (1 << 0));  // Clear Timer 0 interrupt
    }

    if (status & (1 << 1)) {
        // Timer 1 fired
        handle_timer1();
        WRITE(HPET_STATUS, (1 << 1));  // Clear Timer 1 interrupt
    }

    // Clear all pending interrupts at once (alternative approach)
    // WRITE(HPET_STATUS, status);
}
```

2.7 Register Access Conventions

2.7.1 Access Types

Type	Description	Behavior
RO	Read-Only	Software can read, writes ignored
RW	Read-Write	Software can read and write
W1C	Write-1-to-Clear	Write 1 to clear bit, write 0 has no effect
RW/W1C	Read-Write with W1C	Readable, writable, with W1C clear behavior

2.7.2 Reset Values

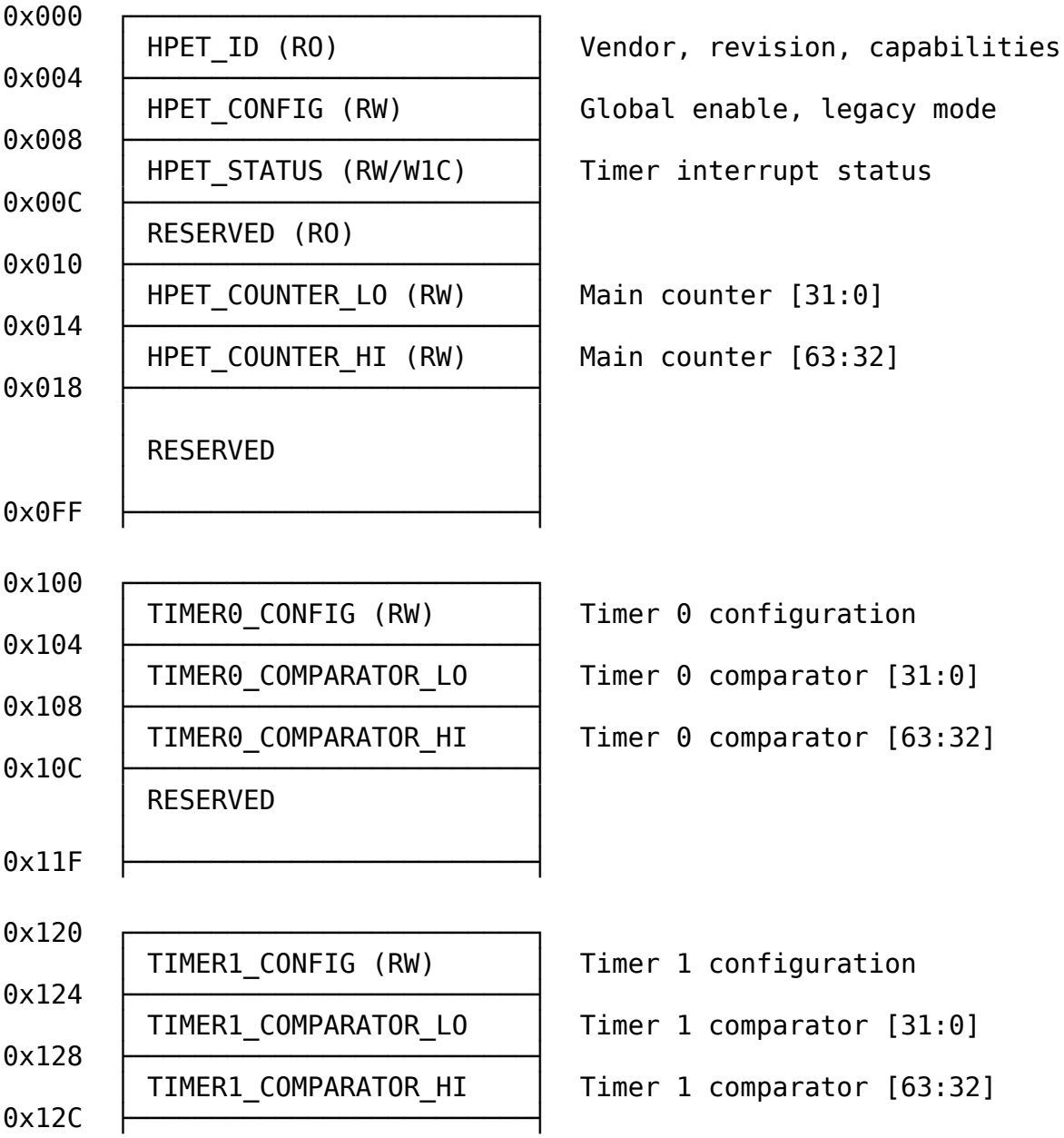
- **Global registers:** Reset to 0x00000000 (except HPET_ID)
- **HPET_ID:** Reset to parameterized values (VENDOR_ID, REVISION_ID, NUM_TIMERS)
- **All timers:** Reset to disabled state (0x00000000)
- **Main counter:** Reset to 0x00000000_00000000

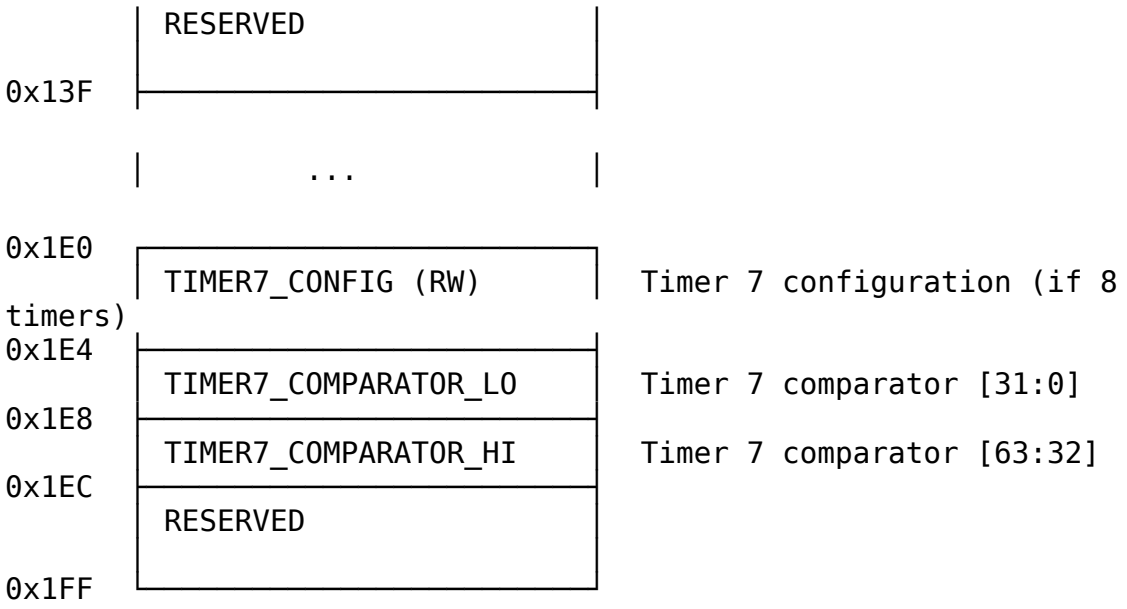
2.7.3 Read/Write Ordering

64-bit Register Writes: 1. Write lower 32 bits (LO) first 2. Write upper 32 bits (HI) second 3. Hardware applies full 64-bit value atomically

64-bit Register Reads: 1. Read lower 32 bits (LO) first 2. Read upper 32 bits (HI) second 3. Be aware of potential rollover during read (rare for slow reads)

2.8 Memory Map Diagram





2.9 Related Documentation

- Chapter 2: Blocks - Block-level architecture
- Chapter 3: Interfaces - Signal interfaces
- Chapter 4: Programming Model - Software usage
- PeakRDL Specification - SystemRDL register definition

2.9.1 Additional Diagrams

- Block Diagram - Top-level architecture
- One-Shot Timer - One-shot mode operation
- Periodic Timer - Periodic mode operation
- Software Init - Initialization sequence
- Interrupt Handling - Interrupt flow
- Timer Mode Switch - Mode switching
- Multi-Timer Concurrent - Concurrent operation
- CDC Handshake - Clock domain crossing

Document Version: 1.0 **Generated:** 2025-10-20 **Based on:** hpet_regs.rdl v2