Unheaded Kingdom
Internet-Draft
Intended status: Experimental
Expires: 20 September 2026

S. Bellis
Unheaded

19 March 2026

Shim Pipeline Specification for the Unheaded Protocol Computer
draft-bellis-unheaded-shim-00

Abstract

This document specifies the Shim pipeline for the Unheaded Protocol Computer (UPC). The Shim translates MBC (Monad Bytecode) programs into eBPF execution contexts, defines the per-hop processing model, and specifies the tick packet protocol that drives distributed computation across IPv6 network hops. The pipeline implements a four-stage architecture: Assembly, Verification, Loading, and Execution, with integrated support for memory-mapped I/O, framebuffer rendering, and CRC validation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them except as reference to a "work in progress."

This Internet-Draft will expire on September 18, 2026.

Copyright Notice

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Shim is the execution engine of the Unheaded Protocol Computer (UPC), responsible for translating and executing Monad Bytecode (MBC) programs within eBPF runtime contexts. It forms the critical bridge between the declarative instruction set defined by the MBC ISA and the native eBPF verifier and execution model of the Linux kernel.

The Shim operates within a four-stage pipeline:

1. Assembly: Text MBC programs are assembled into binary images

2. Verification: Assembled programs are validated against security and conformance rules

3. Loading: Verified programs and supporting data structures are loaded into BPF maps

4. Execution: The fetch-decode-execute cycle runs within XDP context, processing tick packets

This specification defines:

* The binary encoding of MBC programs and instruction formats

* Verification rules enforced before loading and execution

* BPF map structures supporting program state, memory, and I/O

* The tick packet protocol that drives distributed computation

* Memory-mapped I/O semantics for framebuffer and keyboard access

* The Dream Ladder stratification model for conformance levels

1.1. Relationship to Other Specifications

The Shim pipeline operates in conjunction with several related specifications in accordance with [RFC8126]:

* The Protocol Foundation specification defines the Monad wire format (20-byte fixed header carried in IPv6 Hop-by-Hop options per [RFC8200]), which encodes register state and control flags at each hop

* The MBC ISA specification specifies the MBC instruction set architecture (48 opcodes, 16-register architecture, 32-bit words)

* The Sophia Dictionary specification defines dictionary lookup semantics used during MBC execution

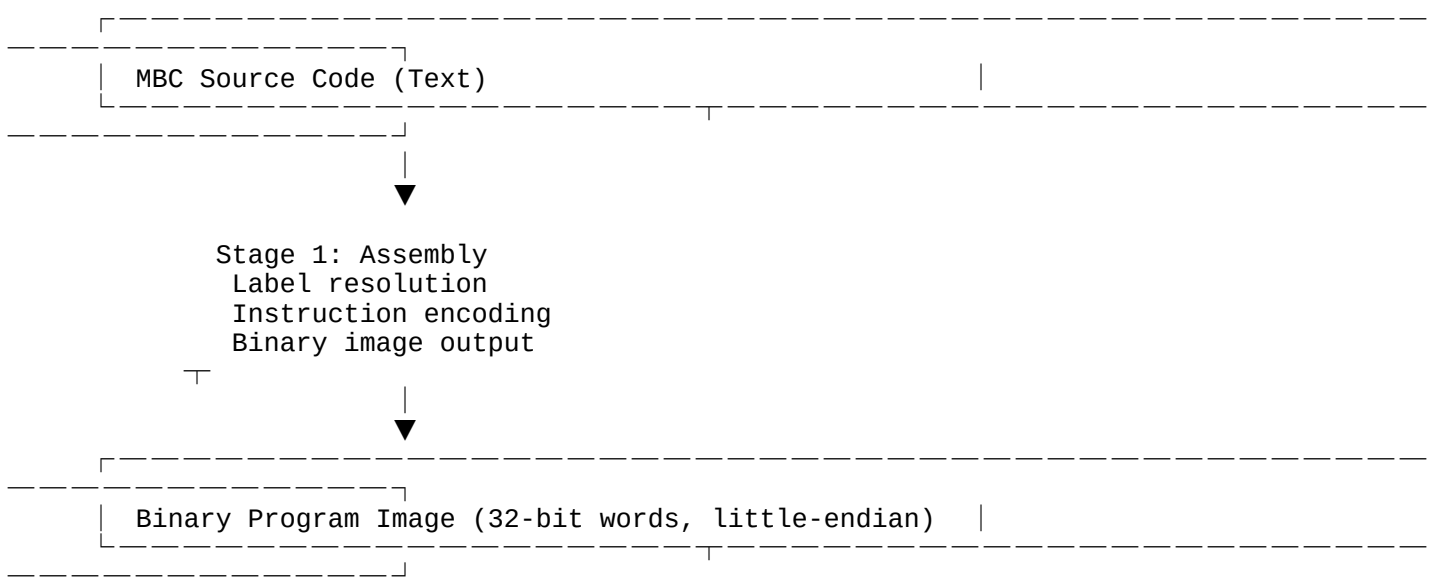
* The Wotan Memory Model specification specifies the BPF map memory model backing the Shim's RAM_MAP and ROM_MAP structures

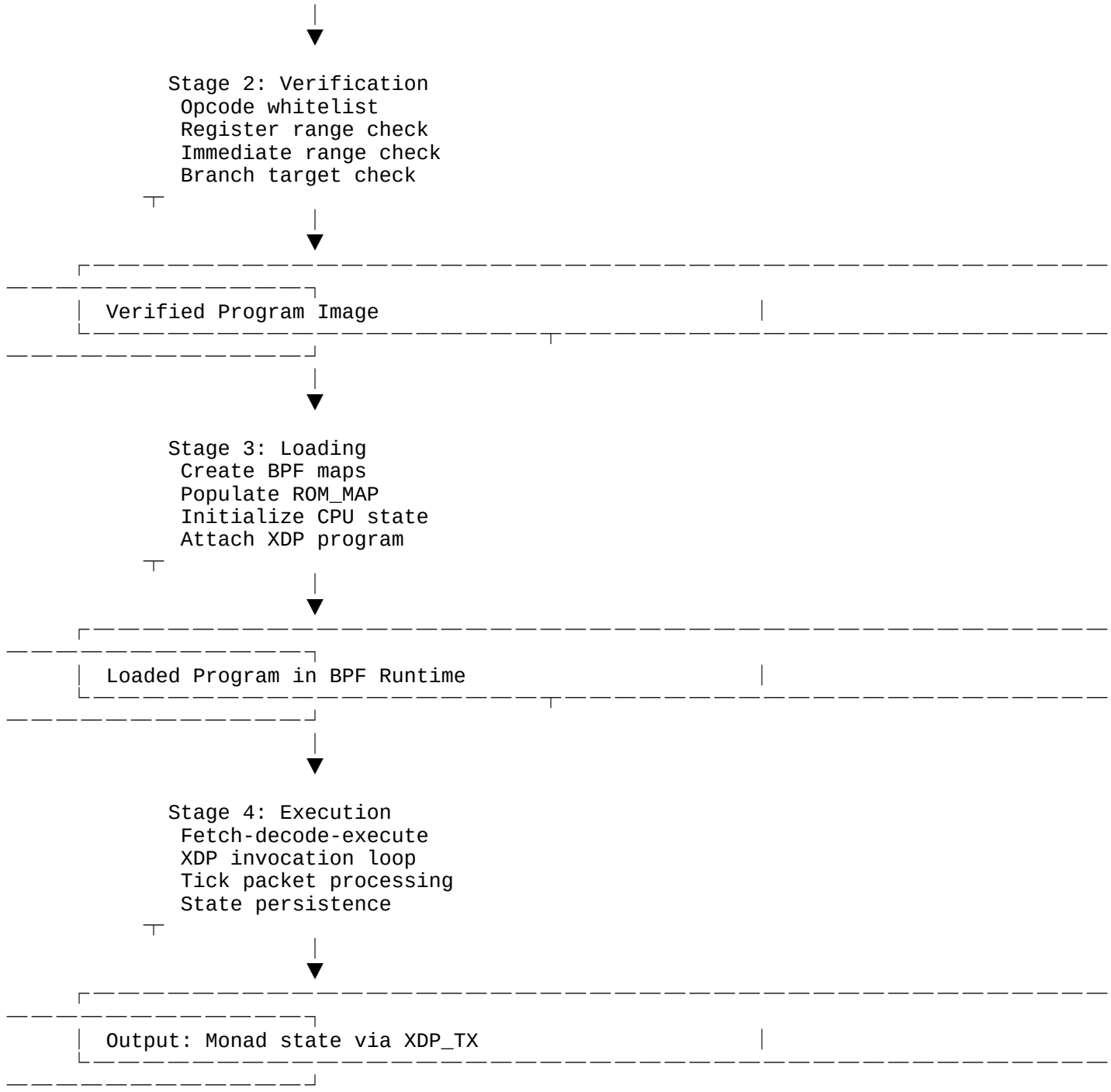
2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Pipeline Overview

The Shim pipeline transforms MBC source code into executable packet processing logic through four sequential stages:





Each stage is atomic and idempotent. Programs rejected at Verification are never loaded. Programs that fail CRC validation during Execution emit an Anomaly event but do not execute MBC.

4. Stage 1: Assembly

Assembly translates MBC instruction mnemonics into a binary program image.

4.1. Input Format

MBC assembly is plain text, one instruction per line. Comments begin with '#' and extend to end of line. Blank lines are ignored. Labels are declared with a trailing colon and must appear alone on a line.

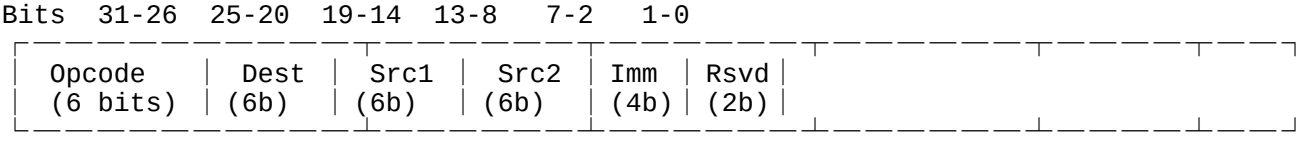
```
# FizzBuzz example
MOVI r0, 1        # Initialize r0 = 1

loop:
ADDI r0, r0, 1    # Increment r0
JNE r0, 100, loop # Jump if not equal

MOV r1, r0        # Move result to r1
HALT              # Stop execution
```

4.2. Binary Encoding

The output is a sequence of 32-bit words in little-endian byte order. Each instruction encodes to one word with format:



Example encoding of MOVI r0, 42:

```
Instruction: MOVI r0, 42
Opcode: 0x0F (MOVI)
Destination: r0 (0)
Immediate: 42 (0x2A)

Binary: 0x0F00002A (little-endian)
Bytes: 2A 00 00 0F
```

4.3. Label Resolution

The assembler performs two passes:

1. First pass: Record label offsets (word addresses in the program image)

2. Second pass: Replace label references in branch instructions with computed offsets

Label references are resolved as relative word offsets from the branch instruction's position. A label at word address N referenced from instruction at word address M becomes offset (N - M).

5. Stage 2: Verification

Verification enforces security and correctness constraints before a program is loaded. Programs MUST pass verification or be rejected entirely.

5.1. Opcode Whitelist

Only 48 opcodes are valid. The verifier MUST reject any program containing an opcode not in the following list:

```
Valid opcodes:
0x00 HALT      0x01 NOP       0x02 MOV       0x03 MOVI
0x04 ADD       0x05 ADDI      0x06 SUB       0x07 SUBI
0x08 MUL       0x09 MULI      0x0A DIV       0x0B DIVI
0x0C MOD       0x0D MODI      0x0E AND       0x0F OR
0x10 XOR       0x11 NOT       0x12 SHL       0x13 SHR
0x14 LD        0x15 LDI       0x16 ST        0x17 STI
0x18 JMP       0x19 JEQ       0x1A JNE       0x1B JLT
0x1C JGT       0x1D CALL      0x1E RET       0x1F PUSH
0x20 POP       0x21 PUSHI     0x22 SYSCALL   0x23 LDMAP
0x24 STMAP     0x25 DICTLOOKUP 0x26 CRC16   0x27 FLAG
0x28 RFLAG     0x29 RESERVED  0x2A RESERVED  0x2B RESERVED
0x2C RESERVED  0x2D RESERVED  ...
```

5.2. Register Range Validation

All register references MUST be in range [0, 15]. The verifier MUST reject programs with register fields containing values greater than 15.

5.3. Immediate Value Range Validation

Immediate values occupy 4 bits in the instruction encoding. MOVI and ADDI instructions with immediate values larger than 15 MUST be rejected at verification time. Programs requiring larger immediates MUST use multi-instruction sequences or load from ROM_MAP.

5.4. Branch Target Validation

Branch instructions (JMP, JEQ, JNE, JLT, JGT) MUST have target offsets that:

* Point to valid instruction boundaries (word-aligned addresses within the program image)
* Do not exceed the program image bounds
* Do not form backward branches with unbounded depth (prevent infinite loops without instruction limit enforcement)

6. Stage 3: Loading

Loading creates the runtime BPF map structures and initializes program state. All maps MUST be created before the program begins execution.

6.1. BPF Map Structures

6.1.1. ROM_MAP

ROM_MAP stores the immutable program image and constant data.

* Type: BPF_MAP_TYPE_ARRAY
* Entries: 262,144 (2^18)
* Value size: 4 bytes per entry
* Total size: 1 MiB
* Access: Read-only during execution

Verified program image is populated into ROM_MAP starting at index 0. Unused entries are zero-filled.

6.1.2. RAM_MAP

RAM_MAP provides the flat address space for data memory and dynamic state.

* Type: BPF_MAP_TYPE_ARRAY
* Entries: 16,777,216 (2^24)
* Value size: 4 bytes per entry

* Total size: 64 MiB

* Access: Read-write during execution

RAM_MAP is initialized to zero. Memory layout is defined in Section 7 (Memory-Mapped I/O).

6.1.3. CPU_MAP

CPU_MAP maintains per-flow CPU state across distributed hops.

* Type: BPF_MAP_TYPE_HASH

* Max entries: 256

* Value size: 128 bytes (MbcCpuState structure)

* Key: Flow tuple (source, destination, flow label)

MbcCpuState structure contains:

* r0-r15: 16 x 32-bit general-purpose registers

* pc: 32-bit program counter (word address)

* sp: 32-bit stack pointer (byte address)

* flags: 16-bit condition code flags

* ticks: 32-bit execution tick counter

* reserved: 8 bytes for future extension

6.1.4. SCREEN_MAP

SCREEN_MAP provides memory-mapped framebuffer access.

* Type: BPF_MAP_TYPE_ARRAY

* Entries: 64,000

* Value size: 1 byte per entry

* Total size: 64 KiB

* Layout: 320x200 pixels, 8-bit palette indices

6.1.5. KBD_MAP

KBD_MAP provides memory-mapped keyboard input.

* Type: BPF_MAP_TYPE_ARRAY
* Entries: 8
* Value size: 4 bytes per entry
* Total size: 32 bytes
* Layout: Bitmask of 256 key states (1 bit per key)

6.1.6. Additional Maps

The following maps support extended functionality:

* TTY_MAP: Terminal I/O ring buffer
* PROC_TABLE: Process/thread state management
* SCHED_STATE: Scheduler state (Level 3+ feature)
* TLB_MAP: Virtual memory translation cache (Level 3+ feature)
* COMPUTE_EVENTS: Event log for anomalies and diagnostics

6.2. Loading Sequence

1. Create all BPF maps in kernel
2. Populate ROM_MAP with verified program image
3. Initialize CPU_MAP with default CPU state (all registers zero, pc=0, sp=0, flags=0)
4. Zero-fill RAM_MAP
5. Attach XDP program to network interface
6. Enable the program for packet processing

7. Stage 4: Execution

Execution implements the fetch-decode-execute cycle within XDP context, processing one instruction per invocation up to a 256-instruction limit.

7.1. Fetch-Decode-Execute Cycle

```
Loop:
1. Fetch: Read instruction from ROM[pc]
2. Decode: Parse opcode and operand fields
3. Execute: Perform operation, update registers/memory
4. Increment: pc += 1
5. Check limit: if ticks >= 256 or HALT, exit loop
6. Else: goto Loop
```

7.2. Instruction Limit and BPF Verifier Compliance

Each XDP invocation MUST execute at most 256 instructions. This limit:

* Prevents CPU exhaustion attacks
* Ensures bounded execution time
* Complies with BPF verifier’s loop detection requirements
* Enables predictable per-hop processing latency

If an instruction limit is reached without HALT, the program suspends and resumes at the next tick packet. PC and register state persist via CPU_MAP across ticks.

7.3. State Persistence

Program state persists across tick packets via BPF maps:

* CPU_MAP: PC, registers, flags persist across ticks
* RAM_MAP: Memory contents persist across ticks
* SCREEN_MAP: Framebuffer contents persist across ticks

State is keyed by flow tuple (source IP, destination IP, flow label). Different flows maintain independent execution contexts.

8. Tick Packet Protocol

Tick packets are IPv6 packets [RFC8200] carrying Monad state and driving distributed computation across hops, with Shim programs verified against the BPF instruction set defined in [RFC9669].

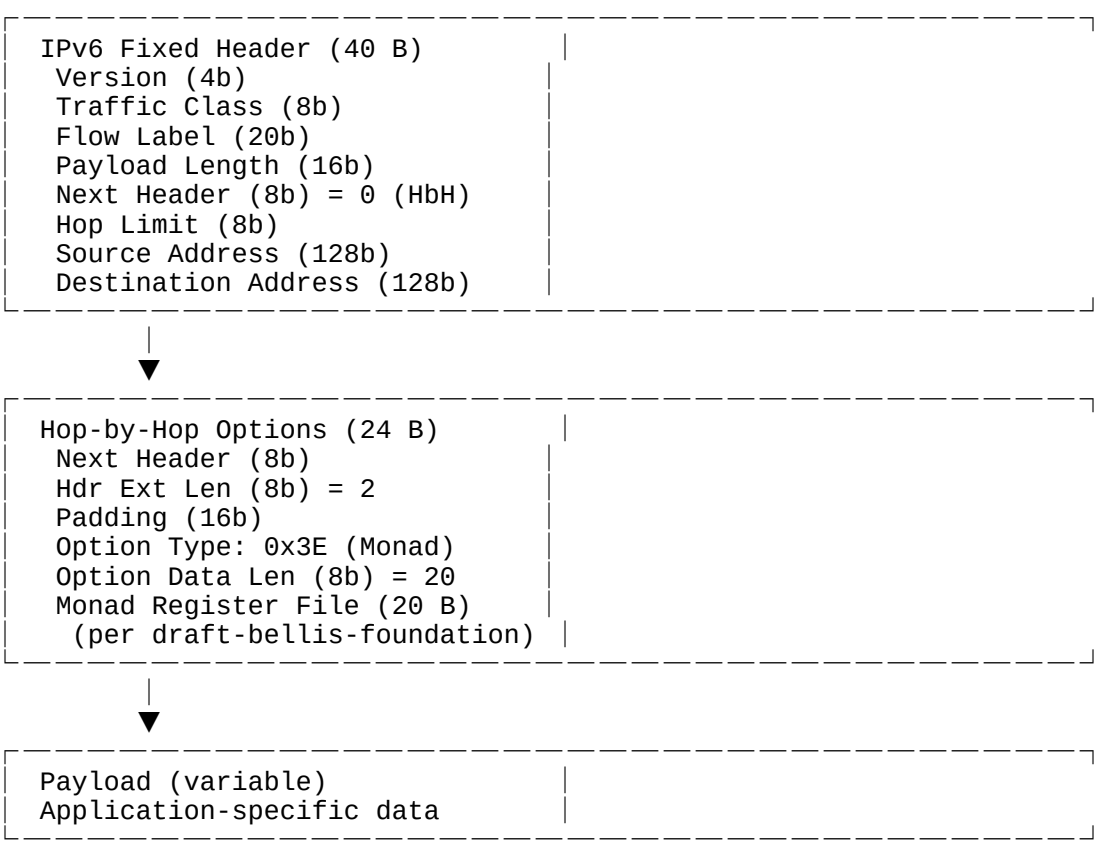
8.1. Packet Structure

A tick packet consists of:

1. IPv6 Fixed Header (40 bytes)

2. Hop-by-Hop Options Header (24 bytes, containing Monad register file)

3. Payload (variable, application-specific data)



8.2. Processing Pipeline

1. Packet arrives at ingress interface

2. XDP program extracts Monad state from HbH option

3. Load/initialize CPU_MAP entry from Monad state

4. Validate Monad CRC-16/CCITT (MUST reject if invalid)

5. Execute MBC program (up to 256 instructions)

6. Recompute Monad CRC-16/CCITT

7. Write updated Monad state to packet HbH option

8. Return XDP_TX (bounce packet to next hop or originating interface)

8.3. Tick Rates

Tick packet generation rates vary by operational mode:

* LOCAL mode (single-hop loopback): 35 Hz (28 ms between ticks)

* DISTRIBUTED mode (multi-hop propagation): ~1 kHz (1 ms between ticks, governed by network propagation)

Tick rate is controlled by application logic, not the Shim itself. The Shim simply processes each arriving tick packet up to its 256-instruction limit.

9. Memory-Mapped I/O

The Shim provides a unified 32-bit flat address space with reserved regions for memory-mapped I/O.

9.1. Physical Address Map

Address Range	Size	Device	Access
0x0000_0000-0x0003_FFFF	256 KB	ROM	R
0x0004_0000-0x0006_7FFF	160 KB	Reserved	-
0x0006_8000-0x0006_8FFF	4 KB	Keyboard (KBD_MAP)	R
0x0006_9000-0x0006_FFFF	28 KB	Reserved	-
0x0007_0000-0x0007_FFFF	64 KB	Framebuffer	R/W
0x0008_0000-0xFFFF_FFFF	4GB -	RAM (RAM_MAP)	R/W

9.2. Keyboard Access (0x0006_8000)

Keyboard state is exposed as an 8-entry array of 32-bit values, where each bit represents one key state (1 = pressed, 0 = released). Total of 256 keys supported.

```
LD r0, 0x68000    # Load keyboard state word 0
LDI r1, 1         # Shift amount for key 1 (ESC)
SHR r0, r0, r1    # Shift right by 1
AND r0, r0, 1     # Mask for single bit
# r0 now contains ESC key state (1 if pressed, 0 if not)
```

9.3. Framebuffer Access (0x0007_0000)

The framebuffer is a 320x200 pixel display with 8-bit palette indices. Pixel (x, y) is stored at byte address 0x70000 + (y * 320) + x.

```
# Write color 42 to pixel (x=100, y=50)
MOVI r0, 100      # x coordinate
MOVI r1, 50       # y coordinate

# Calculate offset: y * 320 + x
MULI r2, r1, 320  # r2 = y * 320
ADD r2, r2, r0    # r2 += x

# Add base address
ADDI r2, r2, 0x70000

# Write color value
MOVI r3, 42       # Color index
ST r2, r3         # Store at calculated address
```

9.4. TTY/Console I/O (0x0007_F000 0x0007_FFFF)

Ring buffer for terminal output. Write operations enqueue characters; read operations drain the queue.

10. Framebuffer Specification

The framebuffer is a 320x200 pixel display with scanline-major layout.

10.1. Dimensions and Layout

* Width: 320 pixels
* Height: 200 scanlines
* Color depth: 8-bit palette index (256 colors)
* Scanline stride: 320 bytes
* Total size: 64 KiB

10.2. Pixel Address Calculation

To write a pixel at coordinate (x, y):

```
address = 0x70000 + (y * 320) + x
```

Constraints:

* 0 ≤ x ≤ 319
* 0 ≤ y ≤ 199
* Out-of-bounds writes are silently dropped by the BPF verifier

10.3. Access Methods

* STB (single-byte write): ST instruction writes one pixel at a time (slowest, most precise)
* SYS_DRAW_FRAME (bulk copy): Syscall that copies entire 64 KiB buffer to framebuffer in one operation (fastest, for scene rendering)

11. Dream Ladder Feature Stratification

The Shim defines conformance levels that implementations must support, organized as a stratified ladder.

11.1. Conformance Levels

11.1.1. Level 0: Microcode (Required)

Foundation layer providing:

* Monad wire format (20-byte HbH option) with CRC-16/CCITT validation
* Sophia dictionary lookups during instruction execution
* XDP packet processing model

Conformance: All implementations MUST support Level 0.

11.1.2. Level 1: Digital (Required)

Instruction execution layer providing:

* Full MBC instruction set (48 opcodes)

* 16-register architecture (r0-r15)
* 32-bit word operations
* Arithmetic, logic, and control flow instructions (no memory operations)
* 256-instruction limit per tick

Conformance: All implementations MUST support Level 1.

11.1.3. Level 2: Mechanical (Required)

Memory operations layer providing:

* RAM_MAP and ROM_MAP (64 MiB + 1 MiB flat address space)
* LD/ST instructions for memory access
* CPU_MAP for state persistence across ticks
* Level 0 and 1 features

Conformance: All implementations MUST support Level 2.

11.1.4. Level 2a: Memory I/O (Recommended)

Memory-mapped I/O layer providing:

* Framebuffer (320x200, 8-bit palette) at 0x70000
* Keyboard input at 0x68000
* TTY console at 0x7F000
* All Level 0-2 features

Conformance: Implementations SHOULD support Level 2a. Level 2a enables interactive applications (games, demos, interactive programs).

11.1.5. Level 3: Interrupts and Exceptions (Optional)

Advanced features for future extension:

* Hardware interrupt handling
* Exception handling and recovery

* Trap vector dispatch

Conformance: Level 3 is OPTIONAL. Implementations may define their own Level 3 features.

11.1.6. Level 4: Scheduling and Multitasking (Optional)

Process and thread management:

* PROC_TABLE for process state

* SCHED_STATE for scheduler decisions

* Preemptive multitasking

Conformance: Level 4 is OPTIONAL.

11.1.7. Level 5+: Virtual Memory, Syscalls, Filesystem (Future)

Reserved for future extension. Currently not defined.

11.2. Conformance Declaration

Implementations MUST declare which levels they support. A conforming implementation MUST support a contiguous range of levels starting from Level 0. For example:

* "Level 0-2": Supports Microcode, Digital, and Mechanical layers

* "Level 0-2a": Supports Mechanical plus Memory I/O

* "Level 0-4": Supports full multitasking stack

An implementation claiming Level N support MUST also support all levels 0 through N-1.

12. CRC Validation Ordering

CRC-16/CCITT validation ensures that register state has not been corrupted during network transmission.

12.1. Pre-Execution Validation

When a tick packet arrives, the Shim MUST:

1. Extract Monad state from IPv6 HbH option

2. Validate CRC-16/CCITT of Monad state

3. If CRC is invalid: emit an Anomaly event to COMPUTE_EVENTS map and DO NOT execute MBC

4. If CRC is valid: proceed to stage 4 execution

This ensures corrupted state never affects program execution.

12.2. Post-Execution Recomputation

After MBC execution completes (whether via HALT or 256-instruction limit), the Shim MUST:

1. Recompute CRC-16/CCITT of the updated register state

2. Write the updated Monad state with new CRC into the outgoing packet’s HbH option

3. Return XDP_TX to transmit the packet

This ensures correct state propagates to the next hop.

12.3. Anomaly Event Format

When CRC validation fails, an Anomaly event is written to COMPUTE_EVENTS with structure:

* timestamp: 64-bit Unix nanoseconds

* event_type: 8-bit code (0x01 = CRC_FAILED)

* flow_tuple: Source IP, Dest IP, Flow Label (for correlation)

* monad_state: Copy of corrupted Monad state

* expected_crc: Expected CRC value

* computed_crc: Computed CRC value

13. IANA Considerations

This specification requests creation of three IANA registries to support interoperability and future standardization.

13.1. Shim Pipeline Stage Registry

Registry Name: Unheaded Shim Pipeline Stages

Range: 0-255

Initial assignments:

* 1: Assembly
* 2: Verification
* 3: Loading
* 4: Execution

13.2. BPF Map Type Registry for UPC

Registry Name: Unheaded BPF Map Types

Range: 0-255

Initial assignments:

* 1: ROM_MAP (read-only program image)
* 2: RAM_MAP (read-write memory)
* 3: CPU_MAP (register state)
* 4: SCREEN_MAP (framebuffer)
* 5: KBD_MAP (keyboard input)
* 6: TTY_MAP (terminal output)
* 7: PROC_TABLE (process state)
* 8: SCHED_STATE (scheduler state)
* 9: TLB_MAP (virtual memory)
* 10: COMPUTE_EVENTS (event log)

13.3. Dream Ladder Level Registry

Registry Name: Unheaded Dream Ladder Levels

Range: 0-7

Initial assignments:

* 0: Microcode (required)

* 1: Digital (required)
* 2: Mechanical (required)
* 2a: Memory I/O (recommended)
* 3: Interrupts and Exceptions (optional)
* 4: Scheduling and Multitasking (optional)
* 5-7: Reserved for future use

14. Security Considerations

14.1. Primary Security Boundary: eBPF Verifier

The eBPF verifier is the primary security boundary. All MBC execution occurs within eBPF context, subject to kernel verification rules following principles established in [RFC9000] for transport reliability:

* Memory access must be in-bounds (BPF map bounds enforce this)
* Loops must be bounded (256-instruction limit enforces this)
* Privilege escalation is not possible (XDP context isolation enforces this)

14.2. CPU Exhaustion Prevention

The 256-instruction limit per tick prevents malicious programs from consuming excessive CPU resources. Programs exceeding this limit are suspended and resume at the next tick. Over a period of N ticks, the maximum total instructions executed is 256 * N, providing predictable resource consumption.

14.3. Memory Access Control

BPF maps provide bounds checking:

* ROM_MAP: Out-of-bounds reads return zero
* RAM_MAP: Out-of-bounds writes are silently dropped
* SCREEN_MAP: Out-of-bounds pixel writes are silently dropped

The kernel enforces all bounds checks; the Shim does not need to replicate this.

14.4. Packet Leakage Prevention

All Shim processing returns XDP_TX (bounce packet to sender or next hop). Return codes XDP_DROP and XDP_PASS are never used, preventing accidental packet leakage into the host stack or transmission to unintended recipients.

14.5. Tick Packet Injection Trust Model

Any node on the network path can inject tick packets. The Shim does not authenticate the origin of tick packets. This is an architectural assumption: use network ACLs and IPSec if authentication is required. The Shim's responsibility is to:

* Validate CRC to detect corruption (but not forgery)

* Process only valid, well-formed MBC instructions

* Prevent execution of out-of-spec opcodes

14.6. Register State Confidentiality

Monad register contents are visible in plaintext at each hop (in the IPv6 HbH option). There is no confidentiality for register state. If confidentiality is required, apply TLS or IPSec encryption at a layer above the Shim.

14.7. Verification as a Security Gate

The Verification stage MUST reject programs with:

* Invalid opcodes (prevents execution of undefined instructions)

* Out-of-range registers (prevents register field corruption)

* Out-of-range immediates (prevents encoding errors)

* Invalid branch targets (prevents control flow attacks)

Programs that fail verification MUST NEVER be loaded. No exceptions.

15. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC9669] Thaler, D., Ed., "BPF Instruction Set Architecture (ISA)", RFC 9669, DOI 10.17487/RFC9669, October 2024, <https://www.rfc-editor.org/info/rfc9669>.

16. Informative References

[RFC9000] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

Author’s Address

Stevie Bellis
Unheaded
Email: stevie@bellis.tech