CTMM Architecture Overview — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

Church Machine Architecture

v1.0 — 2026-04-29 CONFIDENTIAL

Overview

The Church Machine is a capability-secured processor that enforces security at the instruction level. There is no operating system, no privileged mode, no superuser. Every memory access — read or write — passes through a hardware validation gate (mLoad or mSave) that checks an unforgeable Golden Token before permitting the operation.

Design Principles

No Ambient Authority

Traditional systems grant programs implicit access to resources. The Church Machine requires explicit capability tokens for every operation. A program can only access what it holds tokens for.

Domain Purity

The instruction set is split into two domains:

- **Church domain** (10 instructions): Capability manipulation — LOAD, SAVE, CALL, RETURN, CHANGE, SWITCH, TPERM, LAMBDA, ELOADCALL, XLOADLAMBDA. (The 10/10 split is the architectural model; specific implementations may fuse or extend instructions.)
- **Turing domain** (10 instructions + shared RETURN): Data processing — DREAD, DWRITE, BFEXT, BFINS, MCMP, IADD, ISUB, BRANCH, SHL, SHR. (Church Machine

uses ARM-style mnemonics: MOV, ADD, SUB, MUL, DIV, AND, ORR, EOR, LSL, LSR, ASR, CMP, TST, LDI, B, BL.)

A code object (CLOOMC) belongs to the DATA domain — it is data stored in memory, accessed via X permission. Code is never a Church-domain entity. The Church domain handles capabilities (GTs, c-lists); the Turing domain handles computation. A code object may contain Church instructions or Turing instructions, but the object itself is always data. This separation is enforced in hardware.

Abstractions as Security Blocks

An abstraction is a security block — a protected unit of functionality with measurable reliability. Each abstraction has:

- A c-list (CR6 target) containing its capabilities
- Code (CR14 target — CLOOMC) — a DATA-domain object implementing its methods
- Entry via CALL (E-GT); LAMBDA (X-GT) is a method within abstractions, not a separate security block
- MTBF (Mean Time Between Failures) measured by fault reports over time in the namespace

Abstractions are not OS calls — they are namespace entries accessed via Golden Tokens. Every fault against an abstraction is counted and tracked. The abstraction's MTBF is the ratio of uptime to fault count, providing a continuous reliability measure for each security block in the namespace.

Polymorphic Abstraction Interface

Every abstraction — regardless of type or layer — shares the same four structural operations: create, destroy, call, inspect. This uniformity is intentional. The polymorphic interface ensures that creating a math library works the same as creating a hardware driver or a social networking tool. The pattern is repetitive by design.

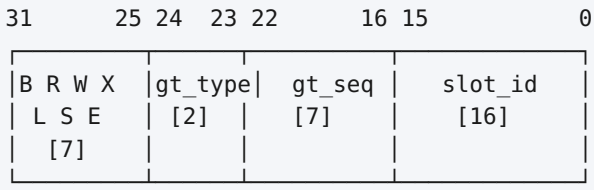
Hardware Device Access (L/S Domain)

All hardware devices (UART, LED, Button, Timer, Display) are accessed through Church domain permissions (L/S/E) — NOT Turing domain (R/W). This enforces capability-gated device access:

- **L (Load)**: Read data from device (receive bytes, read button state, read timer)
- **S (Save)**: Write data to device (send bytes, set LEDs, start timer, write display)
- **E (Enter)**: Call the device abstraction via CALL instruction

R, W, and X permissions are NOT permitted on hardware devices.

Golden Token Format



Bits	Field	Width	Description
[15:0]	`slot_id`	16	Namespace slot ID (0–65,535)
[22:16]	`gt_seq`	7	Revocation sequence counter
[24:23]	`gt_type`	2	GT class (NULL / Inform / Outform / Abstract)
[30:25]	`perms`	6	R, W, X, L, S, E
[31]	`b_flag`	1	Bind flag

gt_seq (7 bits)

Revocation sequence counter. Must match the `gt_seq` stored in NS Entry Word 1 bits [27:21]. On mismatch, the GT is stale — access FAULTs. Revocation is instant: increment the NS entry `gt_seq`, and every outstanding GT referencing that entry dies on next use.

slot_id (16 bits)

Points to a namespace slot. Supports up to 65,536 entries.

Permissions (6 bits)

Bit	Name	Gate	Domain
0	R	DREAD	Turing
1	W	DWRITE	Turing
2	X	LAMBDA	Church
3	L	LOAD	Church
4	S	SAVE	Church
5	E	CALL	Church

R and W are pure Turing permissions (data access). L, S, and E are pure Church permissions (capability access). X (Execute) bridges the two domains: it is grouped with R and W for TPERM domain purity enforcement (presets 3–5: X, RX, RWX), but it gates a Church instruction (LAMBDA) because code application is a capability-mediated operation. A code object is DATA (accessed via X), but applying it is Church's function application. This dual nature is by design — X is the permission that connects the Turing computation domain to the Church security domain.

Type (`gt_type`, bits [24:23])

Value	Type	Meaning
00	NULL	Zero value — no capability. A zeroed GT (gt_type=00) always faults on use.
01	Inform	GT points to memory via an NS entry — abstractions, data objects, lumps
10	Outform	GT references an IDE-managed dependency; lazy-loaded via Locator on first LOAD
11	Abstract	GT IS the value — constants (pi), immutable credentials, PassKey tokens

All abstractions use **Inform (01)** GTs. The Inform GT's `slot_id` indexes a namespace slot that holds the lump base address and limit. CALL loads the lump header from `raw_base` via cLoad and reads `cc` (c-list count) and `n_minus_6` (size exponent) to split the lump into code (CR14, privileged) and c-list (CR6) regions.

Namespace Table Slot Format

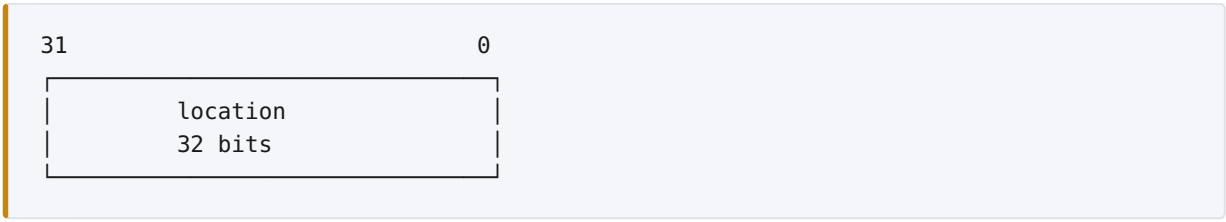
The namespace table begins at `0xFD00`. Each entry occupies exactly **3 consecutive 32-bit words** (12 bytes):

```
NS[slot_id] byte address = 0xFD00 + slot_id × 12
```

The table supports up to **65,536 entries** (bounded by the 16-bit `slot_id` field). Slots 0–7 are reserved by the boot sequence; application abstractions start at slot 8 or higher.

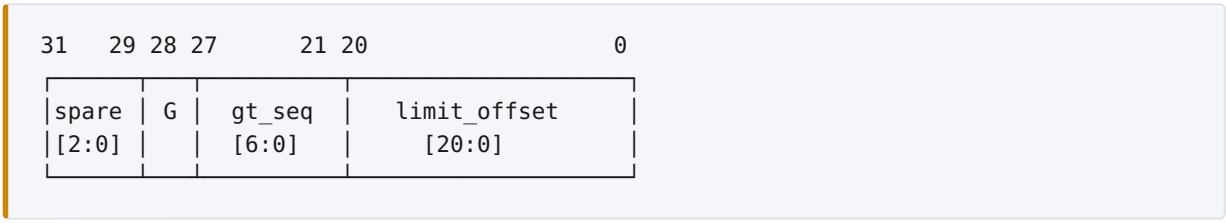
An entry is considered **empty** when both Word 0 and Word 1 are zero.

Word 0 — Location



The base address of the memory object (abstraction lump, data object, or device region) in the unified address space. For an abstraction lump this is where instruction word 0 of the method table lives.

Word 1 — Limit + gt_seq (WORD2_LAYOUT)



Bits	Width	Name	Meaning
[20:0]	21	`limit_offset`	Object size in words minus 1
[27:21]	7	`gt_seq`	Revocation sequence counter; compared against GT `gt_seq` by ChurchNSGate. Increment to revoke all outstanding GTs instantly.
[28]	1	`g_bit`	GC mark bit — may be set by GC; masked before integrity32 check
[31:29]	3	spare	Reserved

Word 2 — integrity32 Check

The 32-bit integrity32 parallel check result, computed over NS Entry Word 0 and Word 1 (with `g_bit` masked to zero before the check).

integrity32 integrity

ChurchNSGate recomputes integrity32 over NS Entry Word 0 and Word 1 (`g_bit` cleared) and compares against NS Entry Word 2. A mismatch faults with `SEAL` error. The covered input includes the base address and limit/gt_seq of the NS entry — the minimum set an attacker would need to forge a valid capability.

gt_seq revocation

Revocation: increment `NS Word 1 [27:21]` by 1. All existing GTs for this entry now have a mismatched `gt_seq` and FAULT on next use. No tracking of outstanding GTs is required — revocation is O(1).

Lump split (abstraction lumps)

When CALL resolves an Inform GT, cLoad reads the **lump header** at `raw_base`. The header encodes: - `cc` — 8-bit c-list count (number of GTs at the top of the lump) - `n_minus_6` — 4-bit size exponent: lump size in words = `2^(n_minus_6 + 6)`

The lump is then divided into two regions:

```
offset 0                          lumpSize-cc        lumpSize
┌──────────────────────────────────┬──────────────────────┐
│  code  (method table + body)     │   c-list (GTs)       │
│  CR14, X-only                    │   CR6, L-only        │
└──────────────────────────────────┴──────────────────────┘

CR14: location = raw_base,              limit = (lumpSize - cc) - 1,  perms = X-only
CR6:  location = raw_base + lumpSize - cc*4,  limit = cc - 1,         perms = L-only
PC   = method_table[method_index] word offset  (method index 0 → word 1; index n →
memory[raw_base + n×4])
```

The c-list count and lump size come from the **lump header** in memory, not from a field in the NS entry.

Complete slot at a glance

```
Offset +0   Word 0 — base      [31:0]   Lump base byte address
Offset +1   Word 1 — limit     [31:29]  spare
            (WORD2_LAYOUT)     [28]     g_bit (GC mark; masked before integrity32)
                               [27:21]  gt_seq (7-bit revocation counter)
                               [20:0]   limit_offset (object size - 1 in words)
Offset +2   Word 2 — integrity [31:0]   integrity32 parallel check
```

Register Architecture

Context Registers (CR0–CR15)

128-bit registers holding Golden Tokens. Each CR stores four 32-bit words (R0–R3):

- **R0**: The GT itself (`b_flag | perms | gt_type | gt_seq | slot_id`)
- **R1**: Lump base address (NS Entry W0)
- **R2**: NS Entry W1 (`spare | g_bit | gt_seq | limit_offset`)
- **R3**: NS Entry W2 (integrity32 parallel check)

> **Convention:** R0–R3 = the 4 words of a Capability Register. W0–W2 = the 3 words of an NS entry.

Special assignments (from `hardware/hw_types.py`): - **CR5**: Heap pointer (CR_HEAP) — bump-allocation frontier - **CR6**: Current capability list (CR_CLIST) — entered via CALL (programmer-accessible) - **CR12**: Thread stack (CR_THRSTK) — privileged, system-wide - **CR13**: Interrupt handler (CR_INTERRUPT) — privileged, system-wide (unchanged by CHANGE) - **CR14**: CLOOMC (CR_CLOOMC) — instruction fetch source, X-only (privileged, per-thread) - **CR15**: Namespace root (CR_NAMESPACE) — privileged, per-thread

Data Registers (DR0–DR15)

32-bit integer registers. DR0 is hardwired to zero.

Flags

ARM-style condition flags: N (negative), Z (zero), C (carry), V (overflow). Set by Turing arithmetic instructions (IADD, ISUB, MCMP). All instructions support conditional execution via 4-bit condition codes.

Memory Architecture

Unified Address Space

The 16-bit physical address space is accessed through the GT gate via mLoad:

```
0x0000 – 0xFCFF    General memory (code + data objects)
0xFD00 – 0xFDFF    Namespace table (NS entries)
0xFE00 – 0xFEFF    Device I/O (UART, LED, Button, Timer, Display) – L/S access only
0xFF00 – 0xFFFF    Machine registers (read-only inspection)
```

MMIO Register Map

All hardware I/O devices are mapped at base address `0x40000000` (bit 30 set, bit 31 clear). The register selector is `addr[5:2]` (4-bit word index within the MMIO range). Per-platform pin assignments and active polarity are in `hardware-tang-nano-20k.md` and `hardware-ti60-f225.md`.

Offset	Address	Device	Register	Description
0	`0x40000000`	LED	LED[0]	LED 0 state — `[2:0]={B,G,R}`
1	`0x40000004`	LED	LED[1]	LED 1 state
2	`0x40000008`	LED	LED[2]	LED 2 state
3	`0x4000000C`	LED	LED[3]	LED 3 state (Tang: drives `led4` pin; `led3` pin is PSRAM CE)
4	`0x40000010`	LED	LED[4]	LED 4 state
5	`0x40000014`	UART	TX	Write byte to transmit
6	`0x40000018`	UART	STATUS	Bit[0]=tx-ready, Bit[1]=rx-ready
7	`0x4000001C`	UART	RX	Read received byte
8	`0x40000020`	Button	BUTTON_STATE	Button bitmask (read-only)
9	`0x40000024`	—	(reserved)	
10	`0x40000028`	Timer	TICKS_LO	Low 32 bits of tick counter
11	`0x4000002C`	Timer	TICKS_HI	High 32 bits of tick counter
12	`0x40000030`	Timer	TOD_EPOCH	Time-of-day epoch
13	`0x40000034`	Timer	ALARM_CMP	Alarm compare register
14	`0x40000038`	Timer	CTL	Timer control — bit[0]=enable

Access via DREAD/DWRITE using an Abstract GT whose `word1_location` falls in the `0xFE000000–0xFEFFFFFF` local peripheral range. The hardware routes `word1_location[7:0]` to the MMIO register selector.

Abstract Address Space (32-bit word1_location)

Abstract GTs (`gt_type = 11₂`) bypass the 16-bit physical map entirely. Their `word1_location` is a 32-bit **Abstract Address** — a hardware-routed sentinel in the IDE-owned reserved range. No real RAM lump can occupy these addresses.

```
0x00000000 – 0xFDFFFFFF    Real RAM – never an Abstract GT address
0xFE000000 – 0xFEFFFFFF    Local hardware peripheral Abstract GTs (UART, GPIO,
Timer…)
0xFF000000                 Home Base tunnel – primary outbound network gateway
0xFF000001 – 0xFF0000FE    IDE-allocated tunnel channels (named remote services)
0xFF0000FF – 0xFFFEFFFF    Reserved for future IDE-defined Abstract resources
0xFFFF0000 – 0xFFFFFFFD    Reserved for future system Abstract GTs
0xFFFFFFFE                 SWITCH PassKey for CR13 (IRQ Thread)
0xFFFFFFFF                 SWITCH PassKey for CR15 (Namespace)
```

See Abstract GT I/O and Network Addressing for the provisioning protocol and security model.

Namespace Entries

Each namespace entry is **3 words (12 bytes)**, stride = `slot_id × 12` from the NS table base:

- **Word 0** (base): 32-bit lump base byte address
- **Word 1** (WORD2_LAYOUT): `spare[31:29] | g_bit[28] | gt_seq[27:21] | limit_offset[20:0]`
- **Word 2** (integrity32 check): 32-bit parallel check over Word 0 and Word 1 (g_bit masked)

The NS table supports up to 65,536 entries (16-bit `slot_id`).

When the GT's `slot_id` identifies an abstraction lump, CALL invokes cLoad, which reads the lump header at `raw_base` to obtain `cc` (c-list count) and `n_minus_6` (size exponent) and performs the lump split. The NS entry itself contains only the base address, gt_seq, and limit_offset — no clistCount or type field.

Integrity = integrity32 recomputed over NS Word 0 and NS Word 1 (`g_bit` masked), compared against NS Word 2 on every ChurchNSGate access. Tamper with any covered field and the check fails — the GT faults on next use.

Boot Sequence

The boot sequence follows a deterministic flow:

1. **FAULT_RST**: All CRs cleared to NULL, all DRs zeroed. M-Elevation ON.
2. **LOAD_NS**: CR15 initialized with GT to Namespace Root (Slot 0).

3. **INIT_THRD**: CR12 initialized with thread stack GT (Slot 1).
4. **INIT_CLIST**: CR6 loaded with Boot C-List (Slot 2).
5. **LOAD_NUC**: CR14 loaded with Boot Code (CLOOMC from Slot 3, privileged). PC = 0.
6. **COMPLETE**: M-Elevation OFF. Machine begins executing boot code.

After boot, the code CALLs Salvation (NS[4]) to verify the security pipeline. Salvation proves LOAD, TPERM, and LAMBDA work correctly, then transitions to Navana (NS[5]). Navana does not RETURN — it becomes the permanent namespace controller, managing all abstractions, intrusion detection (IDS), and system lifecycle indefinitely.

Security Pipeline (mLoad + ChurchNSGate)

Every capability register load passes through this pipeline:

1. **GT Type Check** — `gt_type=00` (NULL) → FAULT immediately
2. **gt_seq Match** — GT `gt_seq` must equal NS Entry Word 1 `gt_seq`
3. **integrity32 Verify** — integrity32 over NS Word 0 and Word 1 (`g_bit` masked) must match NS Entry Word 2
4. **Bounds Check** — Access offset must be within `[0, limit_offset]`
5. **Permission Check** — Required permission bit must be set in GT
6. **G-bit Reset** — NS Entry Word 1 bit [28] `g_bit` cleared (GC liveness proof)
7. **CR Write** — Validated capability written to destination register

mSave (write gate) performs the symmetric check for c-list writes, additionally requiring `B=1` (bit [31] of GT Word 0) on the source GT.

B-bit (Bind)

GT Word 0 bit [31] (`b_flag` in `GT_LAYOUT`). Controls whether a GT can be saved into another c-list:

- B=0 (default): GT cannot be copied to other c-lists — mSave FAULTs
- B=1: GT is bindable — mSave permits the write

The B flag travels with the GT in bit [31] of the 32-bit token word. CALL automatically clears B on all preserved CRs passed to the callee (“no bind by default”). Explicit TPERM with B modifier enables binding.

Instruction Fetch

Instruction fetch uses CR14 (CLOOMC, privileged):

- PC is an offset within the current code object, not an absolute address
- Bounds checked against CR14's limit
- CALL sets CR14 to callee's CLOOMC and PC to the method-table entry (hardware dispatch via imm15; method index 0 → word 1)
- RETURN restores saved CR14 and PC

CALL / RETURN

CALL performs: 1. Validate E permission on target Inform GT (`gt_type=01`) 2. ChurchNSGate validates gt_seq + integrity32 on the NS entry 3. cLoad reads the **lump header** at `raw_base` → extracts `cc` (c-list count, 8-bit) and `n_minus_6` (size exponent, 4-bit) 4. Compute lump split: - `lumpSize = 2^(n_minus_6 + 6)` words - CR14 (code): location = raw_base, limit = (lumpSize - cc) - 1, perms = **X-only** (privileged) - CR6 (c-list): location = raw_base + (lumpSize - cc) × 4, limit = cc - 1, perms = **L-only** 5. Push 2-word call frame: [caller's E-GT | NIA+machine_indicators] 6. Set PC to method-table entry: read `memory[raw_base + method_index × 4]`; zero entry → `FAULT(PRIVATE_METHOD)`; else PC = that word offset. Method index 0 short-circuits to word 1 (lump header at word 0 is never executable).

Frame layout — 2 words only: - Word 0: The caller's own E-GT (the GT that identified the calling abstraction). RETURN uses this to revalidate the caller and re-derive CR6/CR14 via lump split. - Word 1: NIA (return offset into caller's code) | packed machine indicators (LAMBDA-active, condition flags, M-elevation, stackSpace, stackFrames, etc.)

No DRs and no other CRs are pushed. The callee inherits DR0–DR15, CR0–CR5, CR7–CR13, CR15 from the caller unchanged. CR5 (Heap GT) belongs to the thread — it is installed by CHANGE from the incoming thread's Zone ④ bounds and is shared across all abstractions on that thread by software convention.

CR14 and CR6 permissions are architectural invariants — X-only for code, L-only for c-list. The E-GT grants Enter permission to reach the abstraction; CALL enforces the internal domain split. The lump layout places code (method table + instructions) at offset 0, freespace in the middle, and c-list GTs at `lumpSize-cc`. All lumps are allocated as power-of-2 blocks (minimum 64 words, i.e. `n_minus_6=0`).

RETURN: 1. Pop 2-word frame from call stack 2. ChurchNSGate revalidates caller's E-GT (Word 0): gt_seq + integrity32 + G-bit reset (FAULT on failure) 3. cLoad re-runs lump split on caller's lump header → re-derives CR6 (c-list) and CR14 (code) 4. Restore PC from NIA (Word 1) and machine indicators from Word 1

Method Dispatch Modes

CALL supports three method dispatch modes, determined by the instruction's `imm` and `CRd` fields:

Mode	Encoding	Method selector	Use case
Legacy	`CALL CRn` (imm=0)	DR3	Standard call — set DR3 before CALL
C-list indexed	`CALL d, CRs, #imm` (imm≠0, bit 14 clear)	d (0–14)	Direct method select via instruction field
C-list indexed + escape	`CALL 15, CRs, #imm` (d=15)	DR3	Extended method select for >15 methods
Packed	imm bit 14 set	imm[13:8] (6-bit)	Single-instruction operand + dispatch

In the c-list indexed form `CALL d, CRs, #imm`, the first operand `d` is a **method selector** (a plain number 0–15), not a capability register. Only `CRs` (the c-list source) is a capability register.

Escape convention (d=15): When the method selector is 15, the hardware reads DR3 as the extended method selector instead. This allows abstractions with more than 15 methods (such as SlideRule with 22 methods) to be fully addressed. Methods 0–14 use the fast path; method 15 and above use the DR3 escape.

Example — calling SlideRule.Factorial (method index 18) via c-list indexed CALL:

```
IADD  DR3, DR0, #18       ; Method selector: Factorial (index 18)
IADD  DR1, DR0, #10       ; Argument: compute 10!
CALL  15, CR6, #3          ; Load SlideRule from CR6 c-list[3], dispatch via DR3
                           ; Result in DR1
```

Example — calling SlideRule.Multiply (method index 0) directly:

```
IADD  DR1, DR0, #7        ; Left operand
IADD  DR2, DR0, #6        ; Right operand
CALL  0, CR6, #3           ; Load SlideRule from CR6 c-list[3], method 0 = Multiply
                           ; Result in DR1
```

LAMBDA

Lightweight in-scope code application: 1. Validate X permission on target GT 2. Save current PC as lambda return point 3. Execute target code in current scope (no c-list switch) 4. Machine-status fast path: if target code is a single instruction, execute inline

Garbage Collection (PP250)

Deterministic four-phase garbage collection:

1. **Scan** — Walk namespace entries, mark reachable via G-bit
2. **Identify** — Find unreachable entries (G-bit not set)
3. **Clear** — Reclaim unreachable entries
4. **Flip** — Toggle GC polarity for next cycle

PP250 excludes HALT — the machine always returns to boot sequence. Namespace and memory persist across reboots (warm reboot).

Revocation

Revocation is instant, global, and unforgeable:

1. Increment `gt_seq` in NS Entry Word 1 bits [27:21]
2. Every outstanding GT referencing that entry now has a mismatched `gt_seq`
3. Next ChurchNSGate check FAULTs — no need to find or track copies
4. Re-grant by issuing a new GT with the updated `gt_seq` value

Network Transparency

Outform GTs (type=10) with F-bit=1 represent remote resources:

- Access triggers tunnel protocol (HTTPS/RPC)
- Same GT format, same permission model
- mLoad detects F-bit and routes to Tunnel abstraction
- Transparent to application code

Navana as Master Controller

Navana (NS[5]) is the sole namespace entry writer. All NS table modifications go through Navana:

- **Navana.Add**: Find free NS slot, write 3-word entry (base, g_bit+gt_seq+limit_offset, integrity32), return `slot_id` + `gt_seq`
- **Navana.Remove**: Revoke GT (increment `gt_seq` in NS Entry Word 1), free NS slot
- **Navana.Abstraction.Add**: Process compiled abstraction, allocate power-of-2 lump, write lump header (`cc`, `n_minus_6`), write code + c-list GTs, create NS entry, forge E-GT
- **Navana.Abstraction.Update**: Re-carve lump or migrate to larger allocation
- **Navana.Abstraction.Remove**: Revoke GT, free lump, clear NS slot

The one exception: boot writes Navana's own NS entry via mElevation (raw write). After boot, mElevation is dropped and Navana controls all subsequent writes. Mint.Create delegates NS entry creation to Navana.Add.

Loader Mode 1 — Restore (warm-slot eviction/reload)

The **Loader** (NS[19]) manages warm-slot lazy loading. On resource-constrained hardware (Tang Nano 20K, 64 KB BRAM), not all abstraction lumps can be resident simultaneously. The Loader evicts and restores lumps without touching NS entry authority:

- **Eviction**: The entire lump (header + code + c-list) is zeroed. The memory block is freed for alternative use. The NS entry (type, limit, gt_seq, seal) is **never changed** — it remains the live capability reference.
- **Residency signal**: After eviction, `memory[word0_location] == 0`, so lump header `magic = 0x00 ≠ 0x1F`. CALL/LOAD reads this and raises `CODE_NOT_RESIDENT`.
- **Restore**: The Loader writes the full lump (header + code + c-list) at a valid address within the existing NS grant, updates `word0_location`, and recomputes the seal. Type, limit, and gt_seq are never changed — no new authority is minted.

This is distinct from the Outform/Locator protocol (Mode 2), which handles objects that were never instantiated and requires minting a new Inform NS entry from an Abstract capability grant.

Lump Size Minimum

The `n_minus_6` field in the lump header encodes `lumpSize = 2^(n_minus_6 + 6)`. With `n_minus_6 = 0` the minimum representable lump is 64 words — the field has no encoding for a smaller size. The 64-word minimum is therefore self-enforcing by

encoding: hardware can never receive a sub-64-word lump because the encoding cannot represent one. Software and the compiler must allocate at least 64 words (`SLOT_SIZE`).

Upload Format

```
{
  "abstraction": "Name",
  "type": "abstraction",
  "grants": ["E"],
  "capabilities": [{ "target": 7, "name": "Memory", "grants": ["E"] }],
  "methods": [{ "name": "Method", "code": [0x12345678] }]
}
```

Navana.Abstraction.Add validates: `codeSize + cc <= lumpSize`, each capability target exists and creator holds sufficient permissions, `cc <= 255` (8-bit field), `lumpSize` is power-of-2 (minimum 64 words). The lump header (`cc`, `n_minus_6`) is written at offset 0, method table and code words follow, and c-list GTs are placed at `lumpSize - cc`.

CLOOMC++ Compiler

Multi-language compiler targeting Church Machine 20-instruction set:

- **JavaScript front-end** (Phase 1): JS subset → 32-bit code words
- **Haskell front-end** (Phase 1b): Lambda calculus, case expressions, pairs, let bindings → Church Machine instructions

Auto-detection: the compiler identifies the language from source syntax (Haskell uses `method name(args) = expr`, JavaScript uses `method name(args) { ... }`). Both front-ends share the same Resident Object Model and encode back-end.

Resident Object Model

The c-list is the compiler's symbol table for external references. The Resident Object Model maps abstraction names to c-list offsets so that `call(Memory.Allocate(size))` compiles to the correct LOAD offset + CALL sequence. Offsets are generated directly from the upload's capabilities array — the compiler never guesses.

Calling Convention

Registers	Purpose	Saved by
DR0	Hardwired zero	—
DR1-DR3	Arguments / return values	Caller
DR4-DR11	Local variables	Callee
DR12-DR15	Temporaries (compiler scratch)	Caller

Language Mapping

JavaScript constructs map to Church Machine instructions: - `var x = read(addr)` → DREAD - `write(addr, val)` → DWRITE - `x + y` → IADD, `x - y` → ISUB - `if (x == y)` → MCMP + BRANCH.EQ - `call(Abstraction.Method(args))` → LOAD from c-list + CALL - `return(val)` → RETURN - `x << n` → SHL, `x >> n` → SHR - `bitfield(x, pos, width)` → BFEXT / BFINS

Haskell constructs map to Church Machine instructions: - `\x -> body` → LAMBDA (Church numeral encoding, code region refs) - `f x` → CALL / XLOADLAMBDA (function application) - `let x = expr in body` → IADD (register binding) + scope management - `case x of ...` → MCMP + BRANCH chains (pattern matching) - `if c then a else b` → MCMP + conditional BRANCH - `(a, b)` → SHL + BFINS (pair packing into 32-bit word, 16-bit halves) - `fst p` → SHR (extract upper 16 bits) - `snd p` → BFEXT (extract lower 16 bits) - `succ n` → IADD (Church successor) - `pred n` → ISUB (Church predecessor) - `isZero n` → MCMP + conditional IADD - `x + y`, `x - y`, `x * y` → IADD, ISUB, iterative multiply loop - `pure x` → RETURN (monadic return)

Both languages prove the Church Machine is a universal computation target — the same 20 instructions serve as a substrate for imperative and functional paradigms.

Calling Convention

Registers	Purpose	Saved by
DR0	Hardwired zero	—
DR1-DR3	Arguments / return values	Caller
DR4-DR11	Local variables	Callee
DR12-DR15	Temporaries (compiler scratch)	Caller

DR0 is hardwired to zero — it reads as 0 unconditionally after every instruction.

Cross-references

- Lump-Architecture.md — Lump header format, lump split mechanics, and power-of-2 allocation rules
- CM_LUMP_SPECIFICATION.md — Full binary-level lump specification with encoding formulae, example words, and hardware flow diagrams
- foundation-lump-design.md — Boot image design and three-lump foundation architecture (Salvation, Navana, Mint)
- golden-tokens.md — Golden Token format, CRC coverage, permission model, and revocation protocol

---

Confidential — Kenneth Hamer-Hodges — April 2026