Boot Permission Rules — Release 1

Kenneth J Hamer-Hodges

May 2026

Church Machine Boot Permission Rules

v1.0 — 2026-04-29 CONFIDENTIAL

Foundational Principle

The M (Meta/Microcode) permission is a **transient hardware elevation** — set on the CR (register) by microcode, never on the GT (Golden Token) itself. M isolates metadata objects from all regular RWXLSE actions. The GT stored in the namespace carries only the owner-visible permission; the microcode temporarily adds M to the CR during privileged operations.

Context Register Rules

CR15 — Namespace Root

- **GT permission: none (zero RWXLSE)**
- **CR elevation: M only**
- The Namespace is pure metadata. It is not data (no R/W), not code (no X), not a capability container (no L/S/E). M alone grants the microcode access to walk and manage namespace entries. No user instruction can read, write, load, save, or enter the Namespace directly.

CR8 — Thread Identity

- **GT permission: none (zero RWXLSE)**
- **CR elevation: M only**
- The Thread object is pure metadata — it holds the thread’s identity, shadow C-List snippet, and scheduling state. Like the Namespace, it is isolated from all regular permissions. Only microcode (via M) can inspect or update thread state. No user instruction operates on CR8 directly.

CR5 — Services C-List

- **GT permission: L+S**
- **CR elevation: M added by microcode**
- Stable — set at Thread creation by Boot, does not change on CALL/RETURN.
- The Services C-List is the Thread's gateway to its available services. It needs L (Load) to retrieve service capabilities and S (Save) to manage its entries. It contains `self` [E] (the Thread's own abstraction), which in turn contains the Namespace [E] and its methods (Mint, GC, Lookup, etc.). The Thread accesses services via `CALL(Thread.Method(...))`, which internally navigates self → Namespace → Method. The caller never sees the internal structure.

CR6 — Active C-List

- **GT permission: L only**
- **CR elevation: M added by microcode**
- Dynamic — switches on every CALL/RETURN.
- CALL hardcodes CR6 to L-only (Church domain) as an architectural invariant. The GT grants only L (Load), which allows the LOAD instruction to extract capabilities from the C-List into destination CRs via the mLoad validation path. The microcode temporarily elevates M on the CR during LOAD operations for internal access. This enforces the rule that users can only access C-List contents through the controlled mLoad path. No Turing permissions (R, W, X) are permitted on the C-List — domain purity is maintained.
- CR6 contains **symbolic method names** — these are capability entries, not code references. The implementation details of each method are hidden behind the abstraction's nucleus (CR14).

CR14 — Active Nucleus (Method Code)

- **GT permission: X (Execute)**
- **Optional: R if the code region contains constants**
- Dynamic — switches on every CALL/RETURN.
- CR14 holds the currently executing method/code of the active abstraction. X permission allows the processor to fetch and execute instructions from this region. R may be added when the code segment includes inline read-only constants. No L, S, or E — the Nucleus is code, not a capability container.
- CR14 resolves symbolic method names from CR6 into executable code blocks. The dispatch mechanism depends on the abstraction's chosen style: symbolic resolver (high-security), LAMBDA fast-path, or traditional compiled binary. See `docs/dispatch-styles.md`.

The M Elevation Rule

1. M is **never** stored in the GT. It exists only on the CR during microcode execution.
2. The microcode sets M on the CR when it needs to perform a privileged action (e.g., LOAD reads from a C-List, CHANGE updates thread state, namespace walk during GC).
3. M is cleared from the CR when the microcode operation completes.
4. M grants the microcode the ability to perform any action (Load, Save, Read, Write) on the object — but only within the scope of the current microcode operation.
5. No user instruction can set, test, or observe M. It is invisible to the instruction set.

Domain Separation Summary

CR	Object Type	GT Perms	CR Elevation	Stability	Rationale
CR15	Namespace	—	M	Stable	Pure metadata, no user access
CR8	Thread	—	M	Stable	Pure metadata, no user access
CR5	Services C-List	L+S	M (transient)	Stable	Thread’s services gateway, needs Load+Save
CR6	Active C-List	L	M (transient)	Dynamic	Current abstraction’s capability list
CR14	Active Nucleus	X (+R)	—	Dynamic	Current method code, resolves CR6 symbols to code

The architecture defines two mutually exclusive permission domains: **Turing** (R, W, X) for data and code operations, and **Church** (L, S, E) for capability operations through C-Lists and abstraction entry. M is a transient microcode elevation, never stored in the GT. B (Bind) and F (Far/Foreign) are namespace entry metadata, not GT permission bits.

CRC-16/CCITT Integrity Specification

The integrity seal is CRC-16/CCITT (polynomial 0x1021, init 0xFFFF) computed over exactly **89 bits** of input:

```
Input bits (MSB-first):
  gt_word0[24:0]  — 25 bits (gt_type[1:0] | gt_seq[6:0] | object_id[15:0])
  NS Word 0       — 32 bits (base address)
  NS Word 1       — 32 bits (spare[31:28] | gt_seq[27:21] | limit_offset[20:0])
  ──────────────────────────
  Total: 89 bits
```

The permission bits `perms[30:25]` and the bind flag `B[31]` are **excluded** from the CRC input. This allows TPERM to attenuate permissions without requiring a CRC recomputation. The result is stored in NS Entry Word 2 bits [15:0] and recomputed by ChurchNSGate on every access. A mismatch faults with `SEAL` error.

Boot Sequence Permission Flow

1. **Step 1 (Fault Restart)**: Clear all registers. Cold restart.
2. **Step 2 (Load Namespace)**: Microcode writes CR15 with M elevation. GT has zero RWXLSE.
3. **Step 3 (Switch Thread)**: Microcode writes CR8 with M elevation. GT has zero RWXLSE. Also writes CR5 with the Thread's Services C-List (GT has L+S).
4. **Step 4 (Call Boot)**: Microcode writes CR6 (GT has L only, CR gets M during LOAD operations) and CR14 (GT has X, optionally R). NIA set to 0.

Thread Creation via Mint

When Boot creates a Thread (e.g., Kenneth, Matthew, Daniel), it uses `Namespace.Mint(Thread, size, access)`:

1. Boot microcode has M elevation — it can access the Namespace before any Threads exist (the bootstrap chicken-and-egg).
2. `Namespace.Mint` allocates a namespace entry for the new Thread (3-word descriptor: Location, Limit, Seals).
3. Mint computes the MAC, initializes version to 0, assigns the offset.
4. Boot places a Services C-List GT [L+S] in the new Thread's CR5.
5. The Services C-List contains `self` [E] → Namespace [E] → Mint, GC, Lookup, etc.
6. From this point, the Thread can call `CALL(Thread.Mint(type, size, access))` to allocate its own objects.

Implications for LOAD Instruction

When user code executes `LOAD dest src idx`: 1. The instruction handler checks that src CR holds a capability (not NULL). 2. Microcode elevates M on the src CR. 3. With M elevation, microcode performs the internal L (Load) action — reading entry `idx` from the C-List. 4. The loaded capability is placed into the dest CR via mLoad. 5. M is cleared from the src CR. 6. The GT in the src CR still only shows E to the user.

This is the single trusted path: mLoad is the only gate, and M is the key that only microcode holds.

Abstraction Nesting for Mint

The call `CALL(Thread.Mint(type, size, access))` resolves internally as:

```
CR5 (Services C-List) → self (Thread abstraction) → Namespace → Mint(type, size,
access)
```

1. Caller loads `self` from CR5's Services C-List.
2. Thread abstraction checks the thread's resource budget (memory, namespace slots).
3. Thread's microcode loads Mint from the Namespace's C-List (via CR5 → self → Namespace — all hidden from the caller).
4. Namespace.Mint allocates the entry, computes MAC, returns GT in CR0.
5. Thread updates its resource tally.

6. ## Caller receives the new GT in CR0. Never sees the internal plumbing.