

# **CLOOMC Foundation — Architecture Reference**

## **CLOOMC ISA Foundation Document**

**v1.1 — 2026-05-15 CONFIDENTIAL**

This document records the design session held in May 2026 between the original PP250 designer and the Church Machine team. It explains why important decisions were made, not just what the decision might be — so that future followers of this Church Machine movement understand the constraints to respect and the principles to preserve. It starts with the The Six Laws of CLOOMC a one-page summation of the foundational principles of capability-based computer architecture.

1. The Law of Capability where authority flows through unforgeable Golden Tokens
2. The Law of Namespace Privacy where every binding lives inside a Golden Token
3. The Law of Delegation where rights are given, and cannot be seized
4. The Law of Confinement that guarantees computation cannot exceed transparent tokens
5. The Law of Revocation to limit and withdraw any authority previously granted
6. The Law of Integrity using seals to verify granted origin without disclosure

---

## **1. Heritage and Distinction**

### **The First Immersive Capability Computer (PP250)**

The PP250 (Plessey UK, 1972) was the first immersive capability computer to be successfully fielded commercially. It operated for two decades without a single reported security breach to the

capability model. Every object in the PP250 was accessed through a hardware-validated capability key, a descriptor; no program could reach memory it did not hold a descriptor for. The system survived in production and served in the first Gulf War. It accumulated the necessary operational evidence to turn a theoretical model into a proven if incomplete engineering discipline.

The Church Machine is the PP250's direct and architectural complete successor. The lineage is not metaphorical — it is architectural. The key PP250 designer is also after half a century of binary computer failures the Church Machine designer.

## What the Church Machine has Perfected

Three things about flawless computation inherited from the PP250:

1. **The capability model.** Every memory access is mediated by a hardware-validated token. There is no ambient authority; there is no privileged mode that bypasses the check. If you do not hold a valid token, you cannot touch the memory. These laws apply to Boot at the instant power is applied
2. **Hardware-enforced capability keys,** promoted to the digital gold of international cyberspace. The NS table is the direct descendant of the PP250's segment table. Each entry describes a region of memory — its location, its size, and its current version. The hardware recomputes the integrity check on every access and rejects anyentry that has been tampered with.
3. **The principle that capabilities ARE the IDE.** In the PP250 the descriptor table was the system map. In the Church Machine the Namespace table, viewed through the IDE, is the complete, live description of everything the running system can reach. There is no separate registry, no configuration file, no out-of-band channel. The namespace is the system.

## What Is New

Three things are new in the Church Machine:

1. **LUMP architecture.** A LUMP is a power-of-2-sized, self-describing binary package for a single abstraction — its code, its c-list, and its header word in one contiguous block. LUMPs are the packaging and delivery system. They did not exist in the PP250. The PP250 loaded segments from disk; the Church Machine fetches LUMPs from the IDE, from a tunnel, or from the Mum Library. LUMP is how abstractions travel between systems.

2. **CLOOMC ISA.** Capability-Limited / Object-Oriented / Machine-Code is the core technology — the instruction set that runs on the Church Machine processor. CLOOMC is what is compiled, what is deployed, and what the hardware executes. It is not a scripting layer or a bytecode. It is the machine code.
  3. **Golden Token 32-bit encoding.** The PP250's descriptors were wide hardware words. The Church Machine encodes the full capability — slot index, revocation sequence, permission bits, bind flag, and type — into a single 32-bit word. Every GT is a complete, self-contained capability expression that can be validated, forged only with the secret, and revoked in  $O(1)$  by incrementing a 7-bit counter.
- 

## 2. The CLOOMC Capability Model

### Golden Tokens as Symbolic Expressions

A Golden Token is not merely an access key. It is a symbolic expression of functionality. The 32-bit GT word encodes:

- **What** can be accessed (`slot_id`, 16 bits — the namespace index)
- **How** it can be accessed (`dom + perm[2:0]`, 4 bits — domain selector (0=Turing {X,W,R}, 1=Church {E,S,L}) plus 3-bit permission payload; Turing/Church mutual exclusion is structurally enforced by the `dom` bit)
- **Whether** this instance is current (`gt_seq`, 7 bits — revocation counter)
- **What kind** of resource it is (`gt_type`, 2 bits — Null, Inform, Outform, Abstract)
- **Whether** it can be propagated (`b_flag`, 1 bit — bind permission)
- **Whether** it targets a far/remote resource (`f_flag`, 1 bit — Far indicator, per-token)

The permissions are not advisory. The hardware reads the permission bits before permitting any instruction to proceed. A program that holds only an E (Enter) GT for an abstraction cannot read the abstraction's data, cannot write to its c-list, and cannot execute its code directly. It can only call the abstraction's published entry point. This is capability confinement implemented in silicon, not in software policy.

### The DNA Hierarchy

A GT hierarchy is a complete, self-describing blueprint of an application's functional composition. Consider: a thread holds GTs to its abstractions; each abstraction holds GTs to the abstractions

it depends on; those abstractions hold GTs to their dependencies; and so on down to the hardware peripherals. The resulting directed graph is the application's DNA — every function the application can perform is traceable through a chain of validated GTs from the thread's c-list.

This has a consequence that is easy to miss: **you cannot add a capability to a running system without going through the namespace**. There is no back-channel. If a GT does not exist in the chain, the functionality it represents is unreachable. Privilege escalation requires minting a new GT — and minting requires holding Mint's E-GT, which is itself a capability controlled by the namespace.

## **Lambda Calculus Foundation**

Every abstraction is a pure function in the lambda calculus sense: it takes inputs (via its c-list and data registers), produces outputs, and has no side effects beyond what its capabilities explicitly permit. Composition is well-formed by construction: if abstraction A holds an E-GT to abstraction B, then A can call B, and the hardware enforces that the call proceeds exactly as B's interface specifies — no more, no less.

The CLOOMC instruction set is the operational realisation of this model. CALL is function application. RETURN is function completion. LAMBDA is lightweight in-scope application (a function applied within the same capability domain). LOAD and SAVE are c-list read and write — the operations that assemble function compositions.

## **Mathematical Provability**

The type of each token constrains what it can be applied to. An E-GT constrains the holder to CALL only. An R-GT constrains the holder to DREAD only. The hardware enforces these constraints at every instruction boundary. Because the constraints are enforced in hardware and the GT chain is inspectable (via the namespace), the behaviour of the whole system is derivable from the parts: if you can see every GT in the chain and you know the abstraction at each slot, you can predict every operation the system is capable of performing.

This is not just an academic property. It is the basis for the reliability model (Section 3): the capability envelope IS the specification, and deviations from it are detectable faults, not silent corruptions.

## Fail-Safe by Construction

Faults cannot propagate outside the capability envelope. When an abstraction faults — bad GT, bounds violation, permission denied, integrity check failure — the fault is contained at the boundary. The hardware fires the fault handler; the capability chain that led to the fault is preserved in the fault record; no other abstraction's state is disturbed. This is not recovery by hope. It is recovery by hardware geometry.

A fault in abstraction B cannot corrupt abstraction A's c-list because A's c-list is protected by A's lump boundary, which B cannot cross without holding A's S-GT — and A did not give B that GT. Confinement is structural.

## Dynamic Extension

New abstractions can be loaded, new tokens minted, and new capabilities distributed without breaking the proven properties of what is already running. This is possible because the namespace is the authority table: a new entry in the namespace is a new entry — it cannot forge an entry that already exists; it cannot inherit permissions from a neighbouring entry; it is exactly what Mint wrote and nothing else.

---

# 3. The Reliability Model

## The Error Space After Security

Once security is guaranteed — that is, once the capability model is hardware-enforced and the GT chain is the sole path to any resource — the error space for a running system collapses to exactly two categories:

1. **Specification error** — the abstraction was told to do something the designer did not intend.
2. **Implementation error** — the abstraction was asked to do something correct but its code produced the wrong result.

Nothing else is possible. An attacker cannot inject a third category because the capability envelope prevents it. A bug in one abstraction cannot corrupt another because the hardware boundary prevents it. This is why the reliability model can be quantitative.

## Hidden Implementation

Fixing an abstraction is always local. The capability envelope is the contract — the GT defines what the holder can ask for; the hardware enforces it; the lump defines what the abstraction does when asked. As long as the new lump honours the same contract (same entry points, same permission requirements), any holder of the E-GT will see the fix transparently on the next CALL.

**Regression is impossible by construction:** the new lump cannot reach anything the old lump could not reach, because both are confined by the same GT chain.

## The Capability System as Runtime IDE Extension

Every fault carries precise diagnostic information:

- The GT that was being used when the fault occurred
- The permission that was denied (or the check that failed)
- The pipeline stage where the check happened (GT type, gt\_seq, integrity32, bounds, permission)
- The abstraction slot and label
- The instruction mnemonic

This information is not scraped from a stack trace after the fact. It is produced by the hardware pipeline as a structural output of the fault detection mechanism. The IDE captures it. The fault record is as precise as the hardware can make it — which is very precise.

## MTBF Per Abstraction

Every fault event against an abstraction contributes to its MTBF (Mean Time Between Failures) measurement. The MTBF is computed per NS slot: total operational time divided by total fault count. The result is a quantitative reliability measure for every abstraction in the namespace.

Improvement effort is therefore never wasted. The IDE ranks abstractions by MTBF. The weakest link is always visible. A developer looking at the MTBF table knows immediately which abstraction needs attention — not because someone guessed, but because the hardware counted.

## The Closed Feedback Loop

IDE → compile → deploy → fault capture → MTBF ranking → targeted fix → re-deploy

This loop is closed by the capability model. Deployment goes through the namespace (Navana is the sole NS writer). Faults are captured by the hardware pipeline and reported to the IDE via the

call-home mechanism. MTBF is computed server-side from the fault log. The developer sees the MTBF table in the IDE, fixes the weakest abstraction, compiles, and re-deploys. The loop has no gap. Every step is mediated by a capability that the IDE controls.

---

## 4. The Trusted Security Base (TSB)

### The TSB Principle

The Trusted Security Base is the set of components that must be correct for the security model to hold. In the Church Machine, the TSB is defined by a strict rule:

**Only what is logically prior to the first CLOOMC instruction may be in the TSB. Everything else must be a CLOOMC abstraction.**

“Logically prior” means: things the processor needs before it can execute its first instruction. This includes the processor hardware itself (the mLoad pipeline, the GT validation logic, the instruction decoder), the boot ROM that initialises the registers, and the boot image that is present in RAM when power is applied. Nothing else.

### The Irreducible Minimum

The minimum boot image that satisfies the TSB principle contains exactly three things:

1. **One Namespace** — the NS table that describes what physical memory exists and where. Without a namespace, the processor cannot validate any GT. The namespace is logically prior to the first instruction.
2. **One Thread** — the execution context: PC, register file, call stack. Without a thread, there is no execution. The thread lump is logically prior to the first instruction.
3. **One first Abstraction** — the code the thread starts executing. Without a first abstraction, there is nothing to run. The first abstraction is logically prior to the first instruction.

These three together form the **3-LUMP Starter Kit** (Section 7).

## Anything Extra Is a Threat

Every component added to the TSB beyond the irreducible minimum is:

- A complexity cost: more to audit, more to get wrong
- An attack surface: more code running before security is fully established
- A conceptual confusion: something that looks like a CLOOMC abstraction but is not protected by the capability model

The free slot 2 in the current demo boot image (the historical remnant of Startup.Config) is an example of a TSB violation: it sits in the boot image at a fixed address without being either logically prior to the first instruction or a proper CLOOMC abstraction. It fails the TSB test on both counts.

## The LUMP Architecture Must Be Supportive, Not Subtractive

The LUMP architecture (packaging, delivery, lazy load) is the mechanism that allows everything above the irreducible minimum to be a proper CLOOMC abstraction. Navana, Mint, Memory Manager, the Loader, the GC — these are all abstractions delivered as LUMPs, loaded lazily on first CALL. They do not need to be in the TSB. The LUMP architecture is what makes the TSB small enough to actually audit.

---

## 5. Memory Architecture

The memory architecture is defined entirely by the three foundation LUMPs. There are no other configuration parameters.

### Hardware Rules

- **Lump sizes** are powers of 2, minimum 64 words. The mLoad pipeline uses bit-shifts to find lump boundaries — not addition.
- **NS table** is the NS LUMP — it lives at address 0x0000, the start of memory. `totalNamespaceWords` is the programmer's choice, encoded in the NS LUMP header.
- **cc field** (8 bits) limits c-list rows to 255 per lump. It does not limit NS slots — the `GT slot_id` field is 16 bits, allowing up to 65,535 slots.
- **limit17** (17 bits) caps the pool at 131,071 words — enough headroom above the Ti60's 64 KB to make the Ti60 a clean subset, not a tight fit.

## LUMP Types

The `typ` field (bits [9:8] of the header word) identifies one of three LUMP types:

<b>typ</b>	<b>Type</b>	<b>What it defines</b>
00	<b>Abstraction</b>	Executable CLOOMC code body + freespace + GT c-list
01	<b>Namespace object</b> (reserved)	Memory size and namespace size — not yet implemented
10	<b>Thread</b>	Stack size and heap size

## The Three Foundation LUMPs

<b>#</b>	<b>LUMP</b>	<b>NS Slot</b>	<b>Role</b>	<b>Must be in ROM?</b>
1	<b>NS LUMP</b>	0	totalNamespaceWords — the board's physical memory envelope; everything else follows from this one value	Yes — logically prior to everything
2	<b>Thread LUMP</b>	1	Any stack and heap size desired; Thread.CR0 holds the E-GT for the Application LUMP	Yes — logically prior to first instruction
3	<b>Application LUMP</b>	IDE-configured (slot 4 = Salvation on hardware demo)	First abstraction the thread calls via Thread.CR0; content is board- and IDE-dependent	Yes — the entry point

Slot 2 is the first available catalog slot. Slot 3 is the hardware boot code domain (privileged; no user-visible NS table entry). Slot 4 onward is the dynamic pool; the IDE writes the Application LUMP slot into Thread.CR0 via `setBootEntrySlot()`.

## What Follows Automatically

```
foundation_end = NS_LUMP_SIZE + THREAD_LUMP_SIZE
                = 64 + 256 = 320 words = 0x0140 (2-LUMP boot
overhead)
```

Dynamic pool = foundation\_end → totalNamespaceWords - 1

Pool ceiling = totalNamespaceWords - 1  
= 65,535 (Ti60 F225)  
= 131,071 (XC7A100T)

Nothing else needs to be set. The programmer supplies the NS LUMP and Thread LUMP. The hardware boot ROM (3 instructions, see Section 6) handles the rest. Slot 2 is the first available catalog slot; slot 4 onward is the dynamic pool.

---

## 6. The Boot Layout

The DMEM image uses a 4-region layout:

Address	Region	Words	Status
0x0000	NS Lump	64	Necessary – NS root (slot 0)
0x0040	Thread Lump	256	Necessary – boot thread (slot 1)
0x0140	Dynamic Pool	∞	Necessary – allocatable heap
top-0x400	NS Table	1,024	Necessary – capability table

The 3-instruction boot ROM program lives in IMEM (separate from DMEM), starting at byte address 0x0000. It is hardware — not a LUMP, not in the NS table, not user-visible.

NS slot 2 is null (NS entry all-zeros; no physical lump reservation). NS slot 3 is the boot code domain: hardware-privileged, CR14 points here during BOOT\_PROGRAM execution, no user-visible NS table entry. Slot 4 = Salvation (the hardware demo Application LUMP; NUC\_PROGRAM). The first user-authored abstraction is placed at slot 2 onward by the IDE or Navana.

### The 3-Instruction Hardware Boot ROM

The hardware executes exactly three instructions from IMEM on every reset (hardware/boot\_rom.py, BOOT\_PROGRAM):

```
[0] LOAD AL, CR15, CR15[0] – refresh Namespace cap from slot 0 into CR15
[1] CHANGE AL, CR12, CR15, #1 – load Boot.Thread (slot 1); establishes CR0–CR11
[2] CALL AL, CR0, CR0 – enter Thread.CR0 (IDE-configured Application LUMP)
```

This is the complete boot sequence. There is no intermediate state, no privilege escalation, and no code outside the capability model. The IDE configures `Thread.CR0` (via `setBootEntrySlot()`) before power-on; the hardware demo pre-loads it with an E-GT for Salvation (slot 4).

## What Was Removed and Why

**Free slot 2 failed the TSB test.** Slot 2 (formerly `Startup.Config`, address range `0x0140-0x017F`) was dead space with no code, no c-list, and no meaningful lump header. Removing it eliminates 64 words of dead space and gives slot 2 back as the first available catalog slot.

## 7. The 3-LUMP Starter Kit

The clean model. The bitstream has exactly two parts:

**Part 1 — CLOOMC ISA:** the processor hardware. mLoad pipeline, Golden Token validation, instruction execution. This never changes once programmed. Silicon is silicon.

**Part 2 — Read-only RAM image:** exactly three LUMPs.

#	LUMP	NS Slot	Role	Must be in ROM?
1	Namespace LUMP	0	Total physical memory envelope; owned under M authority	Yes — logically prior to everything
2	Thread LUMP	1	Boot execution context; register file, call stack; <code>Thread.CR0</code> = E-GT for the Application LUMP	Yes — logically prior to first instruction
3	Application LUMP	IDE-configured (slot 4 = Salvation on)	First thing the thread calls via <code>Thread.CR0</code> ; board- and	Yes — the entry point

#	LUMP	NS Slot	Role	Must be in ROM?
		hardware demo)	IDE-dependent	

**Boot sequence — three hardware ROM instructions then one CALL:** 1. LOAD AL, CR15, CR15[0] — refresh Namespace cap (slot 0) into CR15. 2. CHANGE AL, CR12, CR15, #1 — load Thread LUMP (slot 1); establishes CR0–CR11 including Thread.CR0. 3. CALL AL, CR0, CR0 — enter Thread.CR0, the IDE-configured Application LUMP.

The IDE writes the chosen slot into Thread.CR0 via `setBootEntrySlot()` before flashing. The hardware demo pre-loads Salvation (slot 4).

**Slot 2 onward is available for catalog abstractions.** Slot 3 is the hardware boot code domain (privileged, no NS table entry). Slot 4 = Salvation (hardware demo Application LUMP). The dynamic pool starts at 0x0140 immediately after Boot.Thread — no gap.

On the **XC7A100T** the Application LUMP is the **Locator** — a CLOOMC abstraction that runs on the FPGA itself, not on the IDE server. The Locator holds three capabilities in its C-list: an Ethernet GT (for network I/O), a Mint GT (for installing fetched LUMPs), and a NamespaceWrite GT (for promoting Outform NS entries to Live). The moment the board is powered, the thread calls the Locator, which opens the Ethernet link to the IDE. Everything else (Memory Manager, Mint, Navana, the full catalogue) arrives via lazy load over that Ethernet connection.

The Ethernet abstraction (NS slot 51) is the transport layer the Locator uses — a separate, lower-level CLOOMC abstraction that wraps the RMII PHY hardware on the QMTECH Wukong board. The Locator calls `Ethernet.Send / Ethernet.Receive`; the Ethernet abstraction talks to the silicon.

On the **Tang Nano 20K** the transport is UART, and the Application LUMP is the UART Locator abstraction — structurally identical to the XC7A100T Locator but holding a UART GT instead of an Ethernet GT.

## Why This Is Correct

The ROM image is not a boot loader. It IS the application in its initial state. The moment power is applied:

- All three LUMPs have valid headers and valid NS entries
- The hardware can validate any GT against the NS table
- The thread can execute its first CALL

There is no intermediate undefined state. There is no moment when the system is “booting” in a way that bypasses the capability model. The capability model is in force from the first clock cycle.

## Everything Else Is Lazy

Above the 3-LUMP foundation, every abstraction is delivered by lazy load: its NS entry is pre-registered in the Namespace LUMP’s c-list (so GTs can be minted against it immediately), but its lump body is fetched from the IDE or the Mum Library on first CALL. The Locator abstraction handles the fetch-inflate-validate-mint sequence transparently. The calling thread sees no difference between a lazy-loaded abstraction and a resident one — only a latency cost.

This means the ROM image can be tiny (3 LUMPs, a few hundred words), and the full system capability is unbounded. The ROM is the security base; the network is the library.

---

## 8. Board Profiles and Pool GT Values

### Comparison Table

Field	Ti60 F225	XC7A100T
totalNamespaceWords	65,536	131,072
foundation_end (NS + Thread only)	0x0140 (320)	0x0140 (320)
Application LUMP base (slot 2)	0x0140 (320)	0x0140 (320)
Pool base (after Application LUMP)	0x0180 (384)	0x0180 (384)
Pool ceiling	65,535 (0xFFFF)	131,071 (0x1FFFF)
limit17 (Memory pool GT)	0x0FFFF	0x1FFFF
Allocatable pool words	~65,087 (~254 KB)	~130,623 (~511 KB)

**Note:** foundation\_end is identical on both boards (NS 64 w + Thread 256 w = 320 words = 0x0140). Null slot 2 has been removed (Task #1205). Lump sizes are programmer choices, not board choices — a programmer using the same sizes on both boards gets the same foundation\_end. The Application LUMP (slot 2) begins immediately at foundation\_end.

## Why `limit17` Matters

`limit17` is the single value in the Memory Manager's pool GT that must change when retargeting to a new board. Everything else is either:

- Hardware-forced (same on all boards): minimum lump size, alignment rules
- Identical by programmer choice: `foundation_end`, lump sizes
- Arithmetically derived: pool base, pool ceiling

The `limit17` value in the pool GT is the upper bound of the dynamic pool: the largest word address the Memory Manager is permitted to allocate from. On the Ti60 it is `0x0FBFF` (64,511). On the XC7A100T it is `0x1FBFF` (130,047). When the programmer retargets from Ti60 to XC7A100T, they update this one value and the Memory Manager immediately sees the larger pool — no other change is required.

This is the practical consequence of the design: because lump sizes are powers of 2 (hardware-forced) and the pool runs to `totalNamespaceWords - 1`, the only variable when retargeting is the programmer's `totalNamespaceWords` choice — and `limit17` is exactly the field that encodes it.

---

## See Also

- [foundation-lump-design.md](#) — Authoritative rules for foundation lump design, programmer-controlled boot image steps, and the IDE role
  - [boot-rom-layout.md](#) — Specific demo boot ROM layout (IMEM map, NUC\_PROGRAM, DEMO\_NAMESPACE, DEMO\_CLIST)
  - [ctmm-memory-map.md](#) — Authoritative CM memory map with NS table, lump headers, and per-board profiles
  - [locator.md](#) — Absent-lump fetch protocol; lazy load lifecycle
  - [architecture.md](#) — Church Machine ISA overview, GT format, register architecture
  - [golden-tokens.md](#) — GT format and MAC rules
  - [namespace-security.md](#) — Namespace integrity model
  - [plan-lazy-load.md](#) — Lazy loading design and Loader abstraction
  - [network-transparency.md](#) — Outform GT network access and RPC tunnel model
-