Mint & PassKey Issuance — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

Mint

v1.0 — 2026-04-30 CONFIDENTIAL

The GT Issuance Mechanism

Status: DRAFT Author: design session 2026-04-30 Depends on: `docs/memory-manager.md §2`, `docs/abstraction-manager.md §4`

---

1. Purpose

Mint is the only mechanism in the Church Turing Machine that may produce a valid Golden Token (GT). It sits just inside the Abstraction Manager (AM) boundary: the AM accepts a cryptographic identity string and decides what capabilities to grant; Mint encodes those capabilities into the 32-bit GT word and returns it as the local session handle.

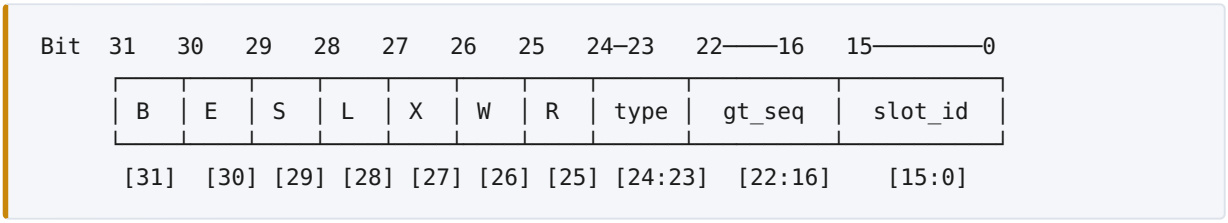
No user-visible code path reaches Mint directly. Every GT a program holds was issued through Mint. Every permission bit the hardware enforces on every instruction was set by Mint at issuance time.

This document is self-standing. A reader who understands the GT bit layout in `docs/memory-manager.md §2` can follow every claim here without reading the rest of the memory manager.

---

2. The GT Word: A Reminder of the Layout

A GT is a single 32-bit word. The hardware checks specific bits on every instruction that names a GT as an operand. The full layout, reproduced from `docs/memory-manager.md §2`, is:



Field	Bits	Width	Meaning
`slot_id`	[15:0]	16	Index into the Namespace table
`gt_seq`	[22:16]	7	Revocation freshness counter; must match NS Entry Word 2
`type`	[24:23]	2	`00`=NULL `01`=Inform `10`=Outform `11`=Abstract
`R`	[25]	1	Read (Turing domain)
`W`	[26]	1	Write (Turing domain)
`X`	[27]	1	Execute (Turing domain)
`L`	[28]	1	Load capability (Church domain)
`S`	[29]	1	Save capability (Church domain)
`E`	[30]	1	Enter — call permission (Church domain)
`B`	[31]	1	Bind — GT may be propagated to another c-list

Mint is the only code path that writes a GT word with a valid `gt_seq`. All other code receives GTs from Mint; it cannot construct a fresh one.

---

3. Public Interface

Mint exposes one primary operation:

```
Encode(base, exp, permsBits, bindable, far) → GT
```

Implementation note: the current simulator stub names this method `Create` (in `simulator/cloomc/mint.cloomc` and `simulator/system_abstractions.js`). The name `Encode` is the canonical specification name. The stub's `Create` and this spec's `Encode` are the same operation; future versions of the implementation are expected to adopt the spec name.

Parameters

Parameter	Type	Maps to GT field
`base`	16-bit unsigned	`slot_id` [15:0]
`exp`	7-bit unsigned	`gt_seq` [22:16]
`permsBits`	6-bit mask (R,W,X,L,S,E)	bits [30:25]
`bindable`	boolean	`B` bit [31]
`far`	boolean	`F` flag in NS Entry Word 1 (lump metadata, not in GT word itself)

`base` is the Namespace slot index that identifies the entity this GT refers to. The hardware uses `slot_id` to look up the entity's physical base address and limit in the Namespace table. Mint receives this index from the Namespace (Navana) after the entity's lump has been registered; it does not choose or guess the slot number.

`exp` is the current value of `gt_seq` for the target slot, read from NS Entry Word 2 at the moment of issuance. Embedding `exp` into the GT word at issuance time is what makes revocation possible: if the Namespace increments `gt_seq` later, every outstanding GT for that slot fails the hardware's freshness check and is immediately invalid. The counter is 7 bits (0–127); it wraps modulo 128 on overflow.

`permsBits` is a 6-bit mask covering the six capability bits R, W, X, L, S, E in that order from LSB. Mint writes these bits directly into GT[30:25]. The hardware checks exactly these bits on every instruction that uses the GT; Mint's encoding is the only opportunity to set them.

`bindable` controls the B bit [31] independently of the six capability bits. Setting B permits the holder to copy this GT into another c-list via `mSave`. Clearing B confines the GT to the c-list it was placed in at issuance. Mint sets B as a separate parameter because the decision of whether a token may propagate is a policy choice made by the AM, not a consequence of any capability combination.

`far` does not appear in the GT word itself. It is a hint written into NS Entry Word 1 that tells the lump loader this entity's backing store may not be resident in local memory and should be fetched from a remote host on first access. Outform GTs always carry `far=1` implicitly. Inform GTs default to `far=0` unless the caller specifies otherwise.

The `type` field

The 2-bit `type` field at GT[24:23] is not a separate parameter to `Encode`. It is a fixed property of the Namespace slot established when the entity's lump was registered via `Navana.Add`. Mint reads the type from the NS entry associated with `base` at the time it performs the NS lookup for `exp`. The valid non-NULL types are:

Value	Name	Meaning
`01`	Inform	Concrete lump, local or lazy-loaded
`10`	Outform	Remote lump, always far-loaded
`11`	Abstract	Value-in-token; no lump behind it

Mint rejects `type = 00` (NULL). See §9 for Abstract GT specifics.

Return value

Mint returns the fully assembled 32-bit GT word:

```
GT = (bindable << 31)
   | (permsBits << 25)        ; bits [30:25]: E S L X W R
   | (type << 23)             ; bits [24:23]: read from NS entry at 'base'
   | (exp << 16)              ; bits [22:16]: gt_seq freshness counter
   | (base & 0xFFFF)          ; bits [15:0]:  slot_id
```

The word is ready for the hardware to check on the first instruction that names it as an operand.

Normative issuance pseudocode

The following pseudocode captures the canonical order of operations for a single GT issuance. Steps 1–3 are caller responsibilities; steps 4–8 are Mint's.

```
1.  size    ← nextPow2(requestedSize)          ; caller: quantise to 2ⁿ words (§8)
2.  loc     ← Memory.Allocate(size)            ; caller: obtain backing lump
3.  (base, exp, type) ← Navana.Add(loc, size, gtType, far)
                                               ; caller: register in Namespace;
                                               ;   base = assigned slot_id
                                               ;   exp  = initial gt_seq (= 0)
                                               ;   type = gtType passed to Navana

4.  assert ¬(Turing(permsBits) ∧ Church(permsBits))  ; DOMAIN_PURITY check
5.  assert ¬(E(permsBits) ∧ (L(permsBits) ∨ S(permsBits)))  ; E_ISOLATION check
6.  assert type ≠ NULL                         ; non-NULL type check
7.  if bindable: permsBits.B ← 1               ; fold B into perm word
8.  GT ← (B(permsBits) << 31) | (permsBits[E:R] << 25) | (type << 23)
           | (exp << 16) | (base & 0xFFFF)
    return GT
```

Steps 4–6 may fault and abort; no GT is returned if any check fails.

---

4. Preconditions and Invariant Checks

Mint enforces two structural invariants before assembling any GT word. Both are checked at mint time, not at use time. A violation raises a fault and returns no GT.

4.1 Domain Purity

The six capability bits divide into two groups:

```
Turing domain:  R (bit 25)   W (bit 26)   X (bit 27)
Church domain:  L (bit 28)   S (bit 29)   E (bit 30)
```

A GT may carry Turing bits or Church bits, but never both. A `permsBits` mask that sets any Turing bit alongside any Church bit raises `DOMAIN_PURITY` and Mint returns no GT.

The rationale is architectural: a region of memory that is readable or writable as data (`R`, `W`) and simultaneously callable as a capability (`L`, `S`, `E`) breaks the hardware's ability to enforce different access rules on different instruction classes. The two domains are physically distinct at the hardware level; Mint enforces the separation at issuance so no mixed token can ever reach user code.

Valid combinations:

```
Turing-only:   R   W   X   RW   RX   WX   RWX
Church-only:   L   S   E   LS
Invalid:       RL  WE  XE  RE   LS+any Turing   (any cross-domain mix)
```

4.2 E Isolation

Within the Church domain, the Enter bit (E) must stand alone.

`LE`, `SE`, and `LSE` are all invalid. An E-GT is the key to invoke an abstraction; an LS-GT is the key to read and write the capability list inside the same abstraction. These must never be the same key. A `permsBits` mask that sets E alongside L or S raises `E_ISOLATION` and Mint returns no GT.

```
Valid Church:    L   S   LS   E
Invalid Church:  LE  SE  LSE
```

E isolation ensures that code which holds only an Enter token cannot inspect or replace the capabilities inside the abstraction it calls.

4.3 Non-NULL type

Mint refuses to issue a GT with `type = 00` (NULL). The NULL GT is the zero word and represents the absence of a capability. It is never constructed by Mint; it appears only as the initial contents of uninitialised c-list slots.

---

5. The `gt_seq` Freshness Counter

5.1 What it is

Every Namespace entry carries a 7-bit `gt_seq` field in NS Entry Word 2, at bits [31:25] of that word. When Mint issues a GT for a given slot, it copies the slot's current `gt_seq` value into GT[22:16]. On every subsequent use of that GT, the hardware extracts `gt_seq` from the GT word, performs a Namespace lookup, and compares the value to the live `gt_seq` in the NS entry. If they differ, the instruction raises a version fault immediately. The GT is dead.

5.2 Where it comes from

Mint reads `gt_seq` from the Namespace at the moment Navana.Add completes. Navana.Add assigns the slot and initialises `gt_seq` to 0 for a newly registered entity. Mint receives `(nsIndex, version)` from Navana.Add and uses `version` directly as `exp` in the Encode call. Mint does not choose the sequence number; the Namespace owns it.

5.3 How it is incremented: Mint.Revoke

Mint exposes a second operation for the system:

```
Revoke(nsIndex) → newSeq
```

Revoke reads NS Entry Word 2 for the given slot, extracts `gt_seq`, increments it modulo 128, and writes it back. It does nothing else. There is no broadcast, no notification, no scan of outstanding GTs.

The effect is immediate and total: every GT that was ever issued for this slot carries the old `gt_seq` value. Every use of any of those GTs will now fail the hardware freshness check. The hardware enforces the revocation; no software sweep is needed. The slot remains valid in the Namespace; a new GT can be issued at any time by calling Encode with the new `gt_seq` value.

The counter wraps. After 127 Revoke calls on a single slot, `gt_seq` returns to 0. If any GT from 128 revocations ago is still held by a program, it will spuriously pass the freshness check. The system must ensure that such long-lived GTs are not held in practice. The 7-bit width is a deliberate hardware constraint; Mint has no mechanism to prevent wrap-around beyond the counter itself.

---

6. Mint.Transfer

Mint exposes one further helper:

```
Transfer(gt, targetCList, targetSlot)
```

Transfer writes a GT word into a specific slot of a specific c-list. This is the only way to place a GT that Mint has just issued into a c-list other than the one Mint itself runs in. No permission bit is changed; the GT is copied verbatim. If the GT has `B=0`, Transfer is still permitted because the copy is performed by Mint — a system actor — not by user code. `B=0` constrains user-level `mSave`; it does not constrain Mint's internal placement.

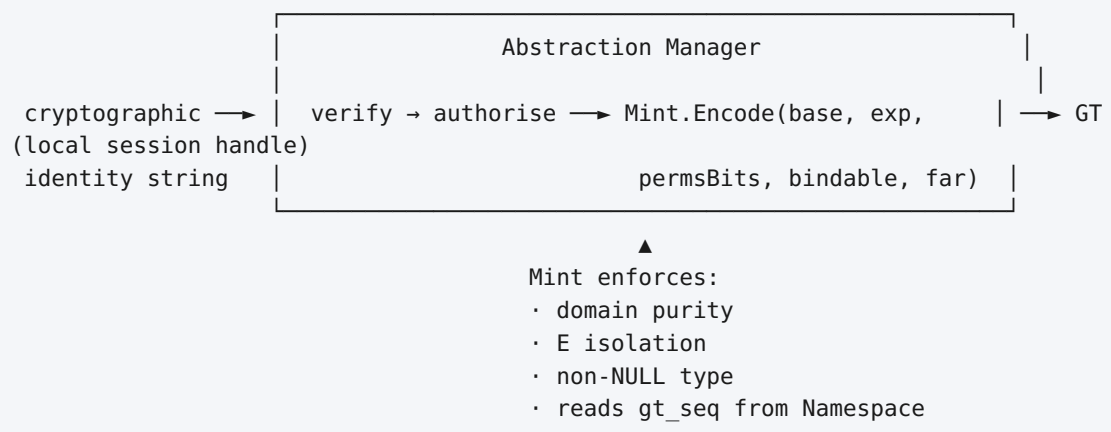
---

7. Mint's Position in the AM Boundary

The AM boundary, from `docs/abstraction-manager.md §4`, is:

```
cryptographic identity string  →  [verify → authorise → issue]  →  local session
handle
```

Mint implements the `issue` step. The AM has already verified the cryptographic string and decided which capabilities the session may receive. It passes the authorised permission set to Mint as `permsBits` and `bindable`. Mint enforces the structural invariants (§4), reads `gt_seq` from the Namespace (§5.2), and returns the assembled GT word. The AM hands that word to the caller as the local session handle.



Mint knows nothing about the cryptographic identity string. It knows nothing about why a particular permission set was authorised. It receives numbers and a type, checks the two invariants, and produces a word. This narrow scope is intentional: capability issuance is a mechanical encoding step, not a policy step. Policy belongs to the AM.

---

8. Relationship to the Namespace

Mint does not allocate Namespace slots directly. The sequence for issuing a GT to a new entity is:

1. **Memory.Allocate(size)** — a backing lump is obtained from the physical pool. Size must be 2^n words ($6 \leq n \leq 14$) per the quantisation rules in `docs/memory-manager.md §3.1`.
2. **Navana.Add(location, limit, gtType, …)** — the lump is registered in the Namespace. Navana assigns a `slot_id` and initialises `gt_seq = 0`. It returns `(nsIndex, version)`.
3. **Mint.Encode(base=nsIndex, exp=version, permsBits, bindable, far)** — the GT word is assembled. The `far` hint is written into NS Entry Word 1 by the caller before or after this step.

Mint is step 3. It has no side-effects on the Namespace or on memory. It reads `gt_seq` (supplied by the caller as `exp`) and writes nothing. The only state change Mint makes in the wider system is through `Mint.Revoke`, which writes a new `gt_seq` value back to one NS entry word.

9. Abstract GT Note

When `type = 11` (Abstract), the GT word carries its value directly rather than through a Namespace lookup. The `slot_id` field is reused as `ab_data` and the `type` subfields carry an `ab_type` discriminant as described in `docs/memory-manager.md §2.2`. Mint.Encode applies the same domain-purity and E-isolation checks to Abstract GTs, but the `far` parameter and the `gt_seq` freshness mechanism are not meaningful for the value-in-token Abstract type. The `exp` field should be set to 0 when encoding an Abstract GT.

10. Summary

Question	Answer
Who may call Mint?	The AM issuance path only. No user-visible E-GT for Mint is ever issued.
What does Mint check?	Domain purity, E isolation, non-NULL type. Nothing else.
Where does `gt_seq` come from?	The Namespace entry, at the moment Navana.Add registers the lump.
How does revocation work?	Mint.Revoke increments `gt_seq` in one NS entry. The hardware rejects all outstanding GTs for that slot.
What does `bindable` control?	Whether the GT holder can propagate the token via `mSave` (B bit [31]).
What does `far` control?	A hint in NS Entry Word 1 — not in the GT word — signalling the lump may not be locally resident.
Does Mint allocate memory?	No. Memory.Allocate and Navana.Add precede Mint. Mint only encodes.

This document describes design intent only. No source files have been modified as a result of this specification.

Confidential — Kenneth Hamer-Hodges — April 2026