Abstract Golden Token — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

Abstract GTs — Self-Describing Device Capabilities

v1.0 — 2026-04-29 CONFIDENTIAL

Introduced in Task #406; UART, Button, and Timer migrated in Task #431.

Overview

An **Abstract GT** is a 32-bit capability word whose entire authority is encoded in the word itself. It needs no namespace table entry and no physical lump in memory — the GT is the capability.

Compare with an Inform GT (type=0b01) which names an NS slot:

Property	Inform GT	Abstract GT
NS table entry	Required	**None**
Physical lump	Required	**None**
Authority storage	NS entry	GT word itself
mLoad required	Yes	**No**
DREAD/DWRITE path	mLoad → mem	Abstract Manager

For 6 LEDs: - **Before (Inform):** 1 NS slot + 1 lump (≥ 64 words) + 6 GTs all pointing at NS slot 12 - **After (Abstract):** 0 NS slots + 0 lump bytes + 6 self-describing GTs

Word Layout

```
 31  30  29  28  27 | 26  25 | 24  23 | 22  ..  16 | 15  ..  0
[   ab_type (5b)   ] [R] [W] [ type ] [  gt_seq  ] [  ab_data  ]
      32 types          I/O   = 0b11     version     16-bit payload
```

Field	Bits	Width	Notes
`ab_type`	[31:27]	5	Abstract category (32 possible)
`R`	[26]	1	Read permission
`W`	[25]	1	Write permission
`type`	[24:23]	2	= `0b11` — identifies Abstract GT
`gt_seq`	[22:16]	7	Version/generation counter
`ab_data`	[15:0]	16	Device-specific payload

X, L, S, E, B are repurposed as `ab_type` bits and must never be treated as permissions on an Abstract GT. `createAbstractGT()` and `create_abstract_gt()` only accept R and W.

Abstract Type Registry (`ab_type`)

`ab_type`	Category	Description
`0x00`	**I/O**	Hardware device pins and registers
`0x01`	**M-Elevation**	Sets CRn(M=1) — the M Abstraction
`0x02–0x1F`	Reserved	Future abstract categories

I/O Sub-Encoding (`ab_type = 0x00`)

```
ab_data[15:8]  =  device_class   (what kind of device)
ab_data[7:0]   =  device_data    (register or pin index within the device)
```

device_class	Device	device_data
0x01	LED	Pin index 0–5
0x02	UART	Register offset: TX=0, STATUS=1, RX=2
0x03	Button	Always 0
0x04	Timer	TICKS_LO=0 ... CTL=4
0x05	Display	DMA register offset

LED example — c-list slots 8–13 (Task #406)

```
Slot 8:  Abstract GT  ab_type=I/O  R|W  LED[0]  → 0x07800100
Slot 9:  Abstract GT  ab_type=I/O  R|W  LED[1]  → 0x07800101
Slot 10: Abstract GT  ab_type=I/O  R|W  LED[2]  → 0x07800102
Slot 11: Abstract GT  ab_type=I/O  R|W  LED[3]  → 0x07800103
Slot 12: Abstract GT  ab_type=I/O  R|W  LED[4]  → 0x07800104
Slot 13: Abstract GT  ab_type=I/O  R|W  LED[5]  → 0x07800105
```

Word breakdown for `0x07800100` (LED[0]): - bits[31:27] = `0b00000` → ab_type = 0x00 (I/O) - bit[26] = `1` → R permission - bit[25] = `1` → W permission - bits[24:23] = `0b11` → type = Abstract - bits[22:16] = `0b0000000` → gt_seq = 0 - bits[15:8] = `0x01` → device_class = LED - bits[7:0] = `0x00` → device_data = pin 0

UART, Button, Timer — c-list slots 14–16 (Task #431)

```
Slot 14: Abstract GT  ab_type=I/O  R|W  UART[0]   → 0x07800200  (TX register)
Slot 15: Abstract GT  ab_type=I/O  R    Button[0]  → 0x05800300  (state register)
Slot 16: Abstract GT  ab_type=I/O  R|W  Timer[0]   → 0x07800400  (TICKS_LO register)
```

NS slots 11 (UART), 13 (Button), 14 (Timer) are freed — no lump, no NS entry. To access a different UART register (e.g. RX), create a new Abstract GT with `device_data=2`: `ab_data = (DEVICE_CLASS_UART << 8) | 2` → GT word `0x07800202`.

DREAD/DWRITE dispatch uses `device_data` to select the register:

```
DREAD  DR1, CR_UART, 0    ; CR_UART holds UART[0] GT
  → Abstract Manager → UART.TX read → DR1 ← uartRegs[0]

DWRITE DR1, CR_TIMER, 0   ; DR1=0xDEAD → write Timer TICKS_LO
  → Abstract Manager → Timer.TICKS_LO write → timerRegs[0] = DR1

DREAD  DR1, CR_BUTTON, 0  ; CR_BUTTON holds Button[0] GT (R only)
  → Abstract Manager → Button.state read → DR1 ← buttonState
```

Instruction Routing

DREAD / DWRITE

When the capability register holds an Abstract GT (type bits = `0b11`), the Abstract Manager intercepts **before** any mLoad call:

```
DREAD  DR1, CR3, 0    ; CR3 holds Abstract LED[0] GT
  → Abstract Manager → _dispatchAbstractDread
  → Read LED[0] state → DR1 ← (this.ledBits >> 0) & 1
  ; No mLoad, no memory access
```

```
DWRITE DR1, CR3, 0    ; DR1=1 → turn LED[0] ON
  → Abstract Manager → _dispatchAbstractDwrite
  → this.ledBits |= (1 << 0)
  ; No mLoad, no memory access
```

Permission checks happen inside the Abstract Manager (R for DREAD, W for DWRITE), not through mLoad.

CALL

CALL on an Abstract I/O GT routes to the **Abstract Manager** via the M-window and dispatches a device method. No mLoad occurs; the GT descriptor is decoded from DR11 (set by `_setMWindow`) before dispatching.

```
CALL CR0              ; CR0 holds Abstract LED[0] GT, DR1[1:0] = method selector
  → Abstract Manager → _dispatchAbstractCall
  → DR1[1:0] = 0b00 (Set)  → ledBits |= (1 << 0); DR1 ← 1 (new state)
  → DR1[1:0] = 0b01 (Clear) → ledBits &= ~(1 << 0); DR1 ← 0 (new state)
  → DR1[1:0] = 0b10 (Toggle) → ledBits ^= (1 << 0); DR1 ← new state
  → DR1[1:0] = 0b11 (State)  → DR1 ← (ledBits >> 0) & 1
  ; No mLoad, no lump access
```

The method selector is encoded in **DR1[1:0]** before the CALL instruction. The result (new LED state: 0=off, 1=on) is written back to DR1 after dispatch. M-window is used internally (CR→DR11–DR15) but is cleared with `writeBack=false` after dispatch — the GT descriptor word itself is never modified by a CALL.

For Abstract GT types other than I/O, or unknown ab_types, CALL faults with `INVALID_OP`.

M Abstraction (`ab_type = 0x01`)

The M Abstraction allows privileged inspection and modification of a 5-word capability register through the DR window. Only CR15 can have M=1. See `.local/tasks/abstract-gt-encoding.md` for the full M API design.

Hardware implementation of the CR→DR M-window is tracked in Task #432.

Helper Functions

JavaScript (simulator.js)

```javascript
// Create an Abstract LED GT for LED pin 2
const ab_data = (ChurchSimulator.DEVICE_CLASS_LED << 8) | 2;
const gt = sim.createAbstractGT(ChurchSimulator.AB_TYPE_IO, {R:1, W:1}, 0, ab_data);

// Parse an Abstract GT word
const info = sim.parseAbstractGT(gt);
// { ab_type: 0, R: 1, W: 1, type: 3, gt_seq: 0, ab_data: 0x0102,
//   device_class: 1, device_data: 2 }
```

Python (server/boot_image.py)

```python
from server.boot_image import (
    create_abstract_gt, AB_TYPE_IO, DEVICE_CLASS_LED
)

ab_data = (DEVICE_CLASS_LED << 8) | 2   # LED pin 2
gt = create_abstract_gt(AB_TYPE_IO, {"R": 1, "W": 1}, 0, ab_data)
# → 0x07800102
```

Format Tag

`BOOT_IMAGE_FORMAT_TAG = 0xB0070431` — bumped from `0xB0070406` (Task #406) when Abstract UART/Button/Timer GTs replaced the Inform GTs in c-list slots 14–16. Both `server/boot_image.py` and `simulator/simulator.js` must agree on this value.

History: - `0xB0070406` (Task #406): Abstract LED GTs (slots 8–13); NS slot 12 freed - `0xB0070431` (Task #431): Abstract UART/Button/Timer GTs (slots 14–16); NS slots 11/13/14 freed

---

CR15 M-Window Hardware Mechanism (Task #432)

Overview

The **M-window** grants a thread momentary write-access to its own namespace CR (CR15). It is implemented entirely in hardware via a 1-bit M-flag latch and a **5-word shadow** (DR11–DR15 / XR11–XR15):

Shadow word	ctmm signal	hardware signal	Content
DR11 / XR11	`m_xr11`	`m_dr11`	Abstract GT word (word0_gt)
DR12 / XR12	`m_xr12`	`m_dr12`	NS entry word0_location
DR13 / XR13	`m_xr13`	`m_dr13`	NS entry word1_authority
DR14 / XR14	`m_xr14`	`m_dr14`	integrity32(DR12, DR13)
DR15 / XR15	`m_xr15`	`m_dr15`	NS entry word3_seals (version + seal)

DR14 / XR14 holds a 32-bit integrity tag computed as `integrity32(location, authority)`. DR15 / XR15 holds the advisory seals word; writeback validates `GT.version == seals.version`.

M-flag lifecycle

Event	M-flag before	Action	M-flag after
M-set (`cr15_m_set`)	0	Copy CR15 → DR11–DR15; set flag	**1**
Valid writeback (`cr15_m_writeback_trigger`, DR11 non-NULL, integrity + version ok)	1	Pack DR11–DR15 → CR15 via `cr_wr`; clear flag	**0**
NULL GT writeback (`cr15_m_writeback_trigger`, DR11 NULL)	1	Raise `INVALID_OP` fault; clear flag	**0**
CHANGE / cross-domain RETURN	1	No FSM trigger; flag preserved	**1**
Global reset (`clear_all`)	any	Wipe all registers	**0**

Signal map

Signal	Direction	Width	Purpose
`cr15_m_set`	in	1	Pulse to copy CR15 shadow + set M-flag
`cr15_m_writeback_trigger`	in	1	Pulse to validate DR11 and initiate writeback
`cr15_m_flag`	out	1	Current M-flag (combinatorial)
`dbg_m_dr11/12/13/14`	out	32×4	Shadow DR reads for test inspection (hardware)
`dbg_m_xr11/12/13/14/15`	out	32×5	Shadow XR reads for test inspection (ctmm)
`dbg_cap_wr_en/addr/data`	in	—	Debug cap-register write port (ctmm testbench)
`m_set_en` (registers)	in	1	Enable CR15 → DR11–DR15 copy + set flag
`m_clear_en` (registers)	in	1	Clear M-flag (no writeback)

FSM states

The M-window FSM lives inside `ChurchCore` / `CTMMCapCore`:

```
IDLE ──(trigger + valid DR11)──→ WRITEBACK ──→ IDLE
     └─(trigger + NULL DR11) ──→ FAULT     ──→ IDLE
```

- **IDLE**: M-set is a combinatorial single-cycle enable (no state change needed).
- **WRITEBACK** (1 cycle): `mwin_cr_wr_en` + `mwin_m_clear_en` → CR15 updated.
- **FAULT** (1 cycle): `mwin_fault_valid` (`INVALID_OP`) + `mwin_m_clear_en`.

`mwin_busy` is HIGH in WRITEBACK and FAULT only, stalling instruction fetch.

Validation (writeback guards)

WRITEBACK applies three checks in order; any failure raises `INVALID_OP` and clears M:

1. **NULL GT**: DR11 `gt_type` field must be non-NULL.
   - ctmm: GT_TYPE_NULL = `0b10`; bits `[1:0]` of XR11.
   - hardware: GT_TYPE_NULL = `0b00`; bits `[24:23]` of DR11.
2. **Integrity**: `integrity32(DR12, DR13) == DR14` (shadow consistency tag).
3. **Version**: `GT.version == seals.version` — `DR11.version` (ctmm bits `[31:25]`) must equal `DR15.version` (ctmm bits `[31:25]`); hardware compares `dr11.gt_seq` vs `dr13.gt_seq`. This revocation check catches writes using a stale M-window from a reclaimed GT slot.

A valid writeback packs DR11–DR14 back into CR15 and clears M.

> **Hardware vs ctmm writeback asymmetry**: The production *`hardware/`* core uses a 3-word *`CAP_REG_LAYOUT`* (word0_gt + word1_location + word2_auth), so DR15 (the seals word) is **advisory-only** in the hardware writeback path — it is used for the version/revocation check but is not written back to *`CR15`*. The *`ctmm_cap_amaranth/`* simulator uses a 4-word *`CAP_REG_LAYOUT`* (adds *`word3_seals`*) and **does** write XR15 back. This is intentional and internally consistent: the hardware validates seals at dispatch time (NS gate) rather than persisting them in the cap register.

Implementation files

File	Role
`ctmm_cap_amaranth/registers.py`	M-flag latch, m_set_en/m_clear_en, m_xr11–15
`ctmm_cap_amaranth/core.py`	M-window FSM, Abstract-GT CALL perm bypass, fault mux, dbg ports
`ctmm_cap_amaranth/call.py`	M_FETCH_NS0–NS3 states, mgt_ns_seals, mgt_set_trigger
`hardware/registers.py`	Production register file: M-flag + DR11–DR15 shadow
`hardware/core.py`	Production M-window FSM, writeback priority in cr_wr mux
`hardware/call.py`	M_FETCH_NS0–NS3 states (hardware), gt_seq revocation
`ctmm_cap_amaranth/testbench.py`	Tests 12A–H: M-set / writeback / CHANGE / fault / real-CALL

Related Tasks

Task	Description
#406	This task — Abstract GT encoding, LED c-list slots 8–13
#430	Amaranth HDL core changes (FPGA synthesis)
#431	UART, Button, Timer as Abstract GTs
#432	CR15 M-window hardware: M-flag latch, 5-word DR11–DR15 shadow, FSM writeback with integrity + version check; M-GT auto-dispatch from CALL via M_FETCH_NS0–NS3

Confidential — Kenneth Hamer-Hodges — April 2026