SB6141 Lab-Target Cousin Mapping Card

2,377 words · ~11 min read

---

*VCA-CSA-101 cross-chapter quick-reference handout. Anchors: + §23 (VCP); arsenal/motorola-sb6141 firmware analysis (2026-04-16); chapter prose §4.3 (BE-32 thread), §5.7 (heterogeneous-MP), §6a (linker / Linux ELF cousin), §8.6.1 (AAPCS thread). *

Purpose: explicit structural-cousin map between every internal component of the **Motorola SB6141 cable modem** (the curriculum's named lab target for `vca-re-101` and `vca-adv-101`) and the corresponding component the student personally builds during CSA-101. **Pedagogical thesis:** *the apparatus you built makes the SB6141 legible.* When `vca-re-101` Lab 8 hands you a packet-processor firmware blob from the SB6141 dump, you will recognise it within minutes. Not because you have RE'd that exact silicon, but because **you have built the structural cousin of every layer the SB6141 contains.**

Print and pin during Ch 4 (BE-32 thread first appears), Ch 5 (heterogeneous-MP introduction), Ch 6a (linker + Linux ELF discussion), Ch 8 (AAPCS calling convention), Ch 12 (Architecture Comparison Sidebar §12.12 walks the table again from the inhabited-pattern angle), then RE-101 Labs 1-8 (where the cousin map is *consulted live*).

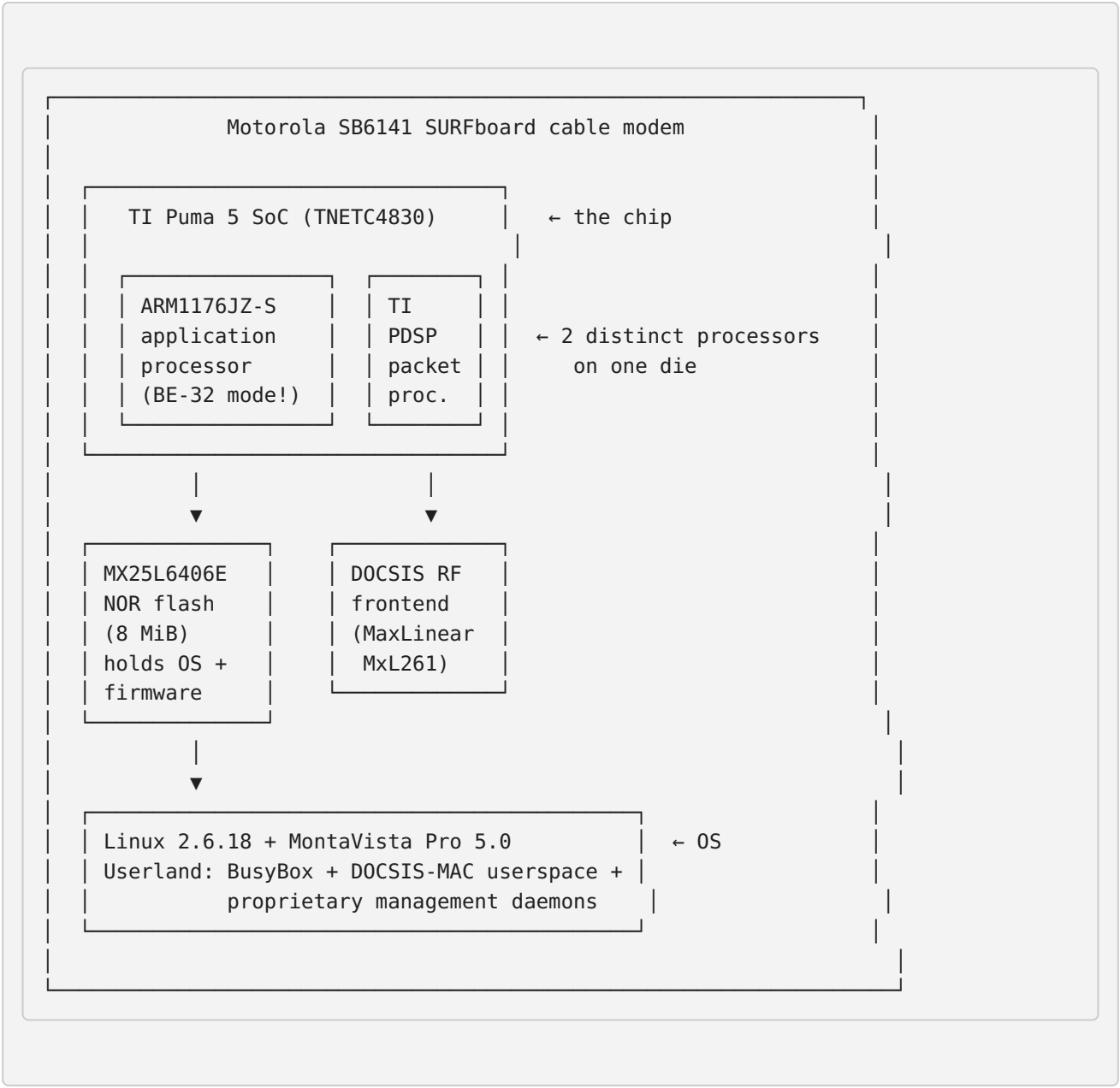
At a glance

Property	Value
Lab target	**Motorola SB6141 SURFboard cable modem** (DOCSIS 3.0; ~2014 release)
Curriculum scope	Named lab target for **RE-101** + **ADV-101**; structural cousin to **CSA-101**
Why this device	Concrete consumer product; available used for ~$10; well-documented hardware; multiple firmware versions; published exploit research
Cousin layers	**6 layers** (silicon / ISA / OS / boot / coprocessor / firmware)
CSA-101 chapter coverage	Ch 4, 5, 6a, 8, 12 explicitly weave the SB6141 thread
arsenal/motorola-sb6141	Vuln-research / exploit-dev project that feeds findings back into curriculum

---

SB6141. Internal architecture summary

The SB6141 is a small embedded system organized around six central components:



The cousin mapping (one row per SB6141 layer)

For each SB6141 component, the structural cousin the student builds during CSA-101, the cousin's CSA-101 chapter, and what that cousin earns the student when they meet the real SB6141 component.

Layer 1: Silicon: the SoC

SB6141 component	CSA-101 cousin	Chapter
TI Puma 5 SoC (TNETC4830). Single die holding application processor + DSP coprocessors + on-chip RAM + memory controller + peripheral interfaces	**Tang Primer 25K (or Tang Nano 20K advanced-track) with Virtus IP Pack**. Single die holding RV32I-Lite CPU + VCP coprocessor + framebuffer + GPIO + system-control region + on-chip BRAM	Ch 5 + Ch 12

What the cousin earns you. *"A SoC is a chip with multiple processors and peripherals on it."* In CSA-101 the abstraction was **inhabited**, the student synthesised the Verilog, watched two processors run concurrently on one die, traced bus traffic with the GAO logic analyzer. Walking into RE-101 Lab 1 ("here is a hex dump of a TNETC4830 SoC"), the student does not need to be told what a SoC *is*; they know because they built one.

Layer 2: ISA: the application processor

SB6141 component	CSA-101 cousin	Chapter
ARM1176JZ-S in BE-32 (big-endian-32) mode, 32-bit ARMv6 architecture, fixed 32-bit instruction width, 16 GP registers, BL+BLR call discipline, AAPCS calling convention, **byte-reversed within each 32-bit word in disassembly**	**Virtus RV32I-Lite**, 32-bit RISC-V subset, fixed 32-bit instruction width, 8 GP registers, jalr+saved-return-address discipline, Virtus calling convention, *little-endian*	Ch 4 + Ch 8

What the cousin earns you. Three things:

1. **ISA literacy.** Both ISAs are RISC-shaped: load/store architecture, fixed instruction width, register-rich. *The student who hand-encoded RV32I-Lite in Ch 4 reads ARM1176 disassembly with one Petzold-style "let me see the structure in hex" lookup table.* Most opcodes look familiar within minutes.
2. **Byte-order surprise.** The SB6141 runs in **BE-32**. Bytes within each 32-bit word are reversed relative to little-endian. *This is the curriculum's deliberate twist*: a bit-pattern that looks like `0xE92D40F0` (little-endian encoding of ARM `push {r4-r7, lr}`) appears in the SB6141 hex dump as `0xF0402DE9`. The Ch 4 §4.3 BE-32 sidebar names this surprise; the Ch 5 §5.7 + Ch 6a §6a-byte-order discussion deepens it; the Ch 8 §8.6.1 AAPCS thread closes it.
3. **Calling convention recognition.** AAPCS uses `r0`-`r3` for arguments + overflow-on-stack, `lr` (link register) for return address, downward-growing stack. **The Virtus VM call protocol from Ch 8 is structurally identical** with two differences: arguments

on stack (not in registers, RV32I-Lite has only 8 registers, no room for argument-passing registers) and upward-growing stack (Nand2Tetris convention; CSA-201 reconciles).

[**Board-work / handout footnote:**] *AAPCS prologue in BE-32 hex looks like:*

```
F0 40 2D E9    push {r4-r7, lr}     ; little-endian: E9 2D 40 F0; meaning: store registers + link to stack
```

In Ch 8 your VM call protocol's caller-side prologue did the structurally identical work in 5-saved-state-words. The shape carries.

Layer 3: Storage: NOR flash

SB6141 component	CSA-101 cousin	Chapter
MX25L6406E NOR flash, 8 MiB. Holds boot ROM + Linux kernel + root filesystem + DOCSIS firmware + persistent config; SPI-attached to the SoC; partitioned into named regions (`u-boot`, `kernel`, `rootfs`, `data`)	**Student-silicon BRAM** (Tang Primer 25K canonical Phase-1 baseline, ~125 KiB; Tang Nano 20K advanced-track, ~64 KiB). Holds OS .text + OS .data + application .text + heap + stack; FPGA-init'd from `$readmemh` flat image; partitioned by linker script into named regions	Ch 3 (memory) + Ch 6a (linker script + section layout) + Ch 12 §12.2 (memory map)

What the cousin earns you. *"Persistent storage on an embedded device is a flash chip with named partitions, mapped at known addresses, holding a sequence of boot artifacts."* The SB6141's flash is 60-128× larger than the student's silicon BRAM (8 MiB vs ~64-125 KiB), but the *organisation principle is identical*. The student who wrote the Ch 6a linker script that places `.text` at one address and `.data` at another, who personally mapped Virtus OS regions to specific BRAM addresses in Ch 12 §12.2, walks into RE-101's "here is the SB6141 flash dump partitioned into u-boot/kernel/rootfs" and recognises **exactly** the same problem at a larger scale. *Naming the partitions, finding the boot loader, identifying the kernel image, mounting the rootfs*, each corresponds to a CSA-101 layer.

The chapter-prose **NOR-flash / BRAM cousin** observation: both are byte-addressable, word-aligned, and word-size-fixed. *The unit-of-knowledge "where do bytes live persistently?" generalizes across consumer firmware.*

Layer 4: OS: Linux 2.6.18 + MontaVista Pro 5.0

SB6141 component	CSA-101 cousin	Chapter
Linux 2.6.18 + MontaVista Pro 5.0. Production embedded Linux: boot loader (u-boot) → kernel → init → BusyBox userland → DOCSIS-MAC userspace + proprietary management daemons. Privileged kernel + user-space split.	**Virtus OS v1**. Flat-image library OS: bootstrap `_start` → library init → `Sys.init` → `Main.main`. Single-task; no privilege boundary; no MMU; ~1,500 source lines	Ch 12 §12.1 (kernel structure styles sidebar) + Ch 12 §12.10 (boot sequence)

What the cousin earns you. Two things:

1. **OS-as-library-of-runtime-services literacy.** *Both Virtus OS v1 and embedded Linux's userspace expose subroutines applications call into.* The Virtus OS's stdlib services (9 primary modules. `Math` / `Memory` / `Output` / `Console` / `Screen` / `GamePad` / `Sound` / `Sys` / `GPIO`. Plus 2 helpers `String` / `Array`; ~34 service entry points per `cross-chapter-stdlib-service-reference.md`) are the structural cousins of `libc`'s `printf`, `malloc`, `strcpy`, `gettimeofday`. **CSA-201 closes the gap**, the student adds privilege boundary + `ecall` trap in CSA-201, at which point Virtus OS v2's structure approaches embedded Linux's privileged-kernel-vs-user-space split.
2. **Boot-sequence recognition.** *On both platforms, control passes through a fixed sequence of layers: power-on → boot ROM → bootloader → kernel/runtime → init → `main`/`Main.main`.* Ch 12 §12.10's instruction-by-instruction bootstrap walk earns the student the right to read `u-boot` source, identify the kernel `start_kernel` entry, and follow the chain to `init` and beyond. *The 9-step Virtus boot sequence is the embedded Linux 9-step boot sequence in pedagogical miniature.*

The chapter-prose Architecture Comparison Sidebar §12.1.2 places **Virtus OS v1 (~1,500 lines)** at the minimum end of the OS-structure curve, with **embedded Linux + busybox (~30M+ lines)** at the monolithic end. *The student inherits, from CSA-101, the position-on-the-curve framework, when they meet a strange OS, they ask "where on the curve?" and the answer locates the system.*

Layer 5: Coprocessor: TI PDSP packet processors

SB6141 component	CSA-101 cousin	Chapter
TI PDSP (Programmable Datapath State Processor). Small microcoded processor running proprietary DOCSIS MAC firmware; runs concurrent with the ARM application processor; communicates via shared SRAM + IRQ line; deterministic real-time (sub-microsecond per-packet processing)	**Virtus Co-Processor (VCP)**. Small 8-bit microcoded processor running audio-PWM firmware; runs concurrent with the RV32I-Lite application CPU; communicates via shared memory + IRQ line; deterministic real-time (deterministic 22 kHz PWM)	Ch 5 §5.7.3 + Ch 12 §12.8 + Ch 12 §12.12

What the cousin earns you. This is the curriculum's **most-deliberately-engineered cousin pair.** *The Ch 12 §12.12 sidebar walks the heterogeneous-multi-processor pattern across TI Sitara/PRU, ESP32/ULP, RP2040/PIO, NXP i.MX RT, Apple M-series, and the **SB6141/PDSP**, and notes that the student's Virtus Console + VCP is the working scale model of the entire family.*

The recognition payoff. When `vca-re-101` Lab 8 hands the student a PDSP firmware blob. A small binary the disassembler does not understand because the PDSP's microcode format is proprietary, the student does not panic. They have built one. They know:

- The coprocessor has its own ISA distinct from the main CPU's.
- It communicates via shared memory + IRQ.
- Its microcode lives in a small ROM/RAM region the boot sequence loads at startup.
- Its job is hard-real-time (bit-banging, packet-processing, DSP) that the application processor cannot do without dedicated silicon.

The cycle-counter measurement from Lab 12.4 Part B (confirming with-audio cycles equal without-audio cycles within measurement noise) *is the single most important experiential demonstration* the cohort makes about this layer. **The audio is free; concurrent processors don't pay for each other's work.** The PDSP is free for the SB6141's ARM in exactly the same way.

Layer 6: Firmware: the application layer

SB6141 component	CSA-101 cousin	Chapter
Proprietary DOCSIS firmware + management daemons, the application code that runs on top of Linux, written in C, compiled with GCC, linked into ELF binaries living in the rootfs partition; calls into libc, calls into kernel via syscalls, reads from /dev nodes for hardware access	**Lab 12.5 Virtus Console capstone**, the student's HLL application that runs on top of Virtus OS v1, written in `.virtus` source, compiled with the student's compiler, linked into a flat image; calls into the 14-service stdlib, calls into the OS via direct `jalr`, reads from peripheral memory addresses for hardware access	Ch 11 + Ch 12 §12.5 (Lab 12.5)

What the cousin earns you. *The application-on-OS pattern is universal.* The SB6141's DOCSIS daemon is a binary built by some compiler from source the student does not have, running on an OS the student did not build, against hardware the student did not design, but **every layer of that stack has a CSA-101 structural cousin the student personally wrote.** When RE-101 Lab 6 hands the student a stripped binary from the SB6141's userland and asks "what does this program do?", the student recognises:

- Compiler-emitted prologue/epilogue patterns (Ch 10 + Ch 11 lecture-notes catalogues these in detail; Lab 11 Ghidra-on-compiler-output is the explicit precursor).
- Symbol-table erasure (Ch 10 weave #1 + §10.4 "the four-step namespace cascade". Names the programmer wrote got erased four times).
- Library-call sequences (Ch 11 §11.3. `call <Module>.<method> <argc>` translates to a `jalr` to the OS-resident entry point).
- Stack-frame discipline (Ch 8's call protocol in pedagogical miniature; AAPCS in production).

The cognitive distance from "I see a firmware partition in the rootfs" to "I understand what's running there" *collapses* because the student personally wrote the smaller version of every piece they will encounter. *That collapse is the curriculum's most-central pedagogical artifact.*

Summary table, all 6 layers at a glance

SB6141 layer	SB6141 specifics	CSA-101 cousin	Chapter
Silicon (SoC)	TI Puma 5 (TNETC4830)	Tang Primer 25K (canonical) or Tang Nano 20K (advanced-track) + Virtus IP Pack	Ch 5, 12
ISA (app proc)	ARM1176JZ-S, BE-32	RV32I-Lite, little-endian	Ch 4, 8
Storage	MX25L6406E NOR flash, 8 MiB	Tang BRAM (~125 KiB Primer 25K / ~64 KiB Nano 20K)	Ch 3, 6a, 12
OS	Linux 2.6.18 + MontaVista Pro 5.0	Virtus OS v1 (library OS)	Ch 12
Coprocessor	TI PDSP packet processors	Virtus Co-Processor (VCP)	Ch 5, 12
Firmware	Proprietary DOCSIS + mgmt daemons	Lab 12.5 capstone	Ch 11, 12

What the SB6141 has that CSA-101 doesn't (forward-pointers)

The cousin map is *not* one-to-one across every dimension. Some SB6141 layers correspond to material CSA-101 doesn't cover, and CSA-201 / RE-101 / RE-201 / advanced electives close those gaps.

SB6141 has	Closed in
Privilege boundary (Linux user/kernel split, ARMv6 mode register)	CSA-201 §2 (privilege levels + `ecall`)
MMU + paged virtual memory	CSA-201 §7 (Sv32 paging)
Filesystem (ext2 rootfs on flash)	CSA-201 (external-DRAM + SD-card track)
Network stack (DOCSIS MAC + IP + DNS)	NET-101 + CSA-201
Multiple userland processes + scheduler	CSA-201 §6 (preemption + scheduler)
Memory-safety mitigations (W^X, ASLR, stack canaries, CFI, Linux ships these by 2014)	CSA-201 §8 + XD strand
RF analog frontend (MaxLinear MxL261)	con-101 (RF/SDR work; future) + WIR-101
Cryptographic engines (AES, hash)	(advanced electives; sec-101 covers algorithms; con-101 covers acceleration)

The student should not arrive at RE-101 expecting CSA-101 to have prepared them for every SB6141 surface. *The forward-pointers above are explicit;* the student takes CSA-201 / RE-201 / advanced electives in parallel as they encounter SB6141 surfaces CSA-101 didn't cover.

RE-101 lab-by-lab usage of this card

The SB6141 cousin map is consulted *live* during RE-101 labs. Approximate per-lab application:

RE-101 lab	What the cousin map provides
Lab 1 *Teardown + flash dump*	Storage cousin (Layer 3). *the flash is partitioned the way Virtus's linker script partitioned BRAM*
Lab 2 *Boot ROM + u-boot identification*	OS cousin (Layer 4). *the boot sequence is the Ch 12 §12.10 walked-by-instruction sequence at a larger scale*
Lab 3 *Linux kernel image extraction*	OS cousin (Layer 4). *kernel image format is ELF; the student already knows ELF cousin to VOF from Ch 6a*
Lab 4 *ELF header reading + symbol table*	Storage + OS cousin. *the readelf / ELF discussion from CSA-101's Toolchain Diary lands here*
Lab 5 *ARM1176 disassembly + AAPCS recognition*	ISA cousin (Layer 2). *Ch 4 + Ch 8's AAPCS forward-pointer; BE-32 byte-reversal discipline*
Lab 6 *Userland binary identification + library-call recognition*	Firmware cousin (Layer 6). *the Ch 11 + Ch 12 stdlib-call patterns map directly to libc / Linux syscalls*
Lab 7 *Cross-references + dataflow*	Firmware cousin. *Ghidra-at-system-scale from Ch 12 toolchain diary*
Lab 8 *PDSP firmware blob*	Coprocessor cousin (Layer 5). *the VCP's microcode + shared-memory protocol is the structural template*
Final capstone *Cohesive teardown report*	All 6 layers. *the cousin map is the report's spine*

The card is itself a deliverable in RE-101's grading rubric: students annotate it with *their* SB6141-specific findings during each lab.

Where to read more

- **Course overviews** (`website/vca-re-101.html`, `website/vca-adv-101.html`). Lab-target alignment context.
- **VCP architecture** (`cross-chapter-stdlib-service-reference.md`). The coprocessor cousin.
- **`arsenal/motorola-sb6141/`**. Vuln-research / exploit-dev work on the SB6141; pedagogically-defensible findings flow back into curriculum.
- **Chapter prose**. Specific weaves:
  - Ch 4 §4.3 (BE-32 introduction)

  - Ch 5 §5.7.3 (heterogeneous-MP sidebar, the SB6141 first appears in the table)
  - Ch 6a (Linux ELF cousin)
  - Ch 8 §8.6.1 (AAPCS thread + byte-order surprise walked)
  - Ch 12 §12.12 (heterogeneous-MP sidebar inhabited; SB6141 in the canonical table)
- **`vca-re-101` curriculum** (separate repo at `cybersecurity/academy/vca-re-101-reverse-engineering`), the RE course that consumes this map directly.
