The Last Resort: Debugging Embedded Systems with Unconventional Methods

Pass The SALT 2025

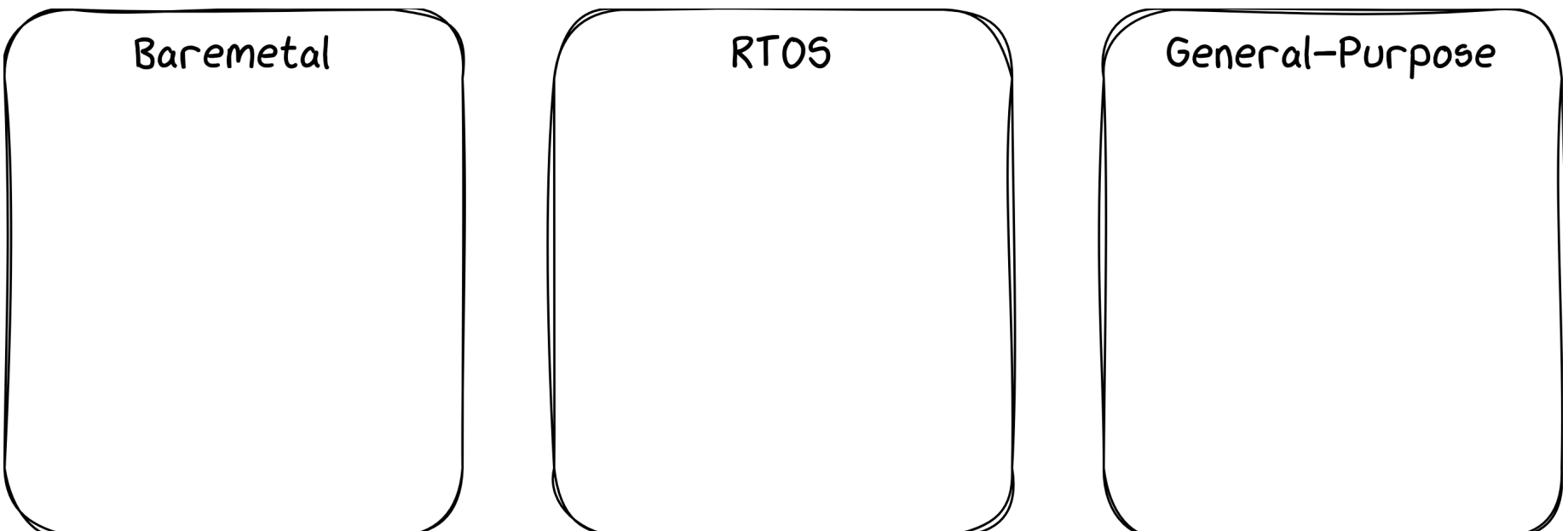
Vincent Lopes

01/07/2025

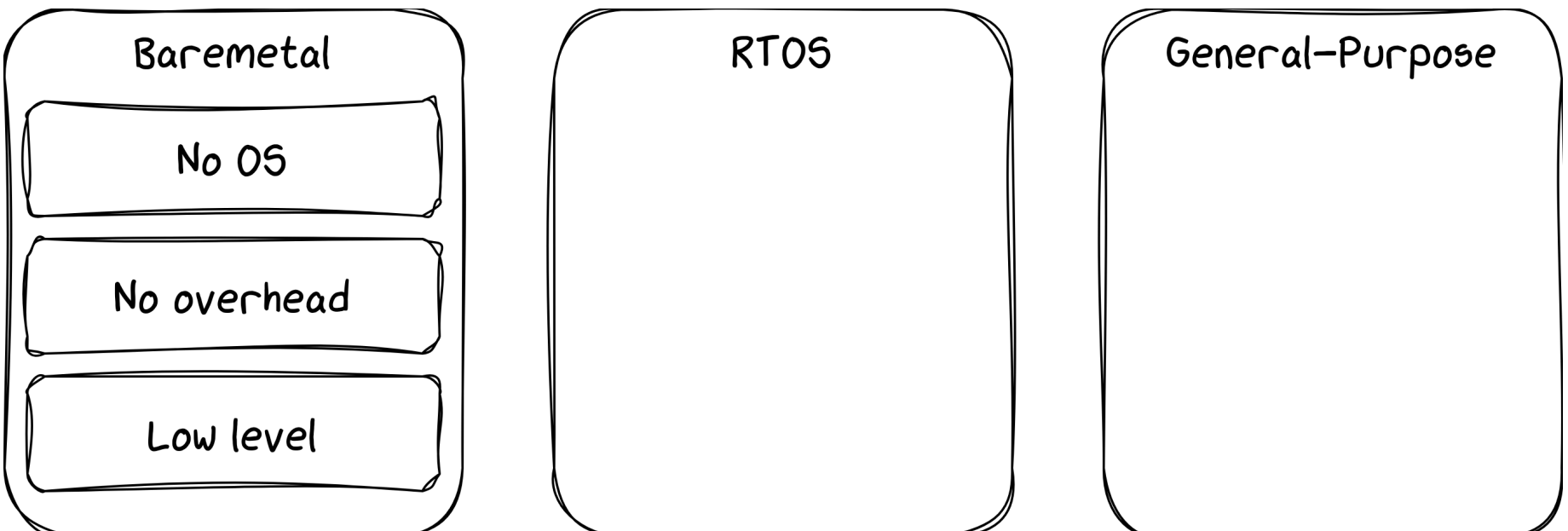
Quarkslab

Context

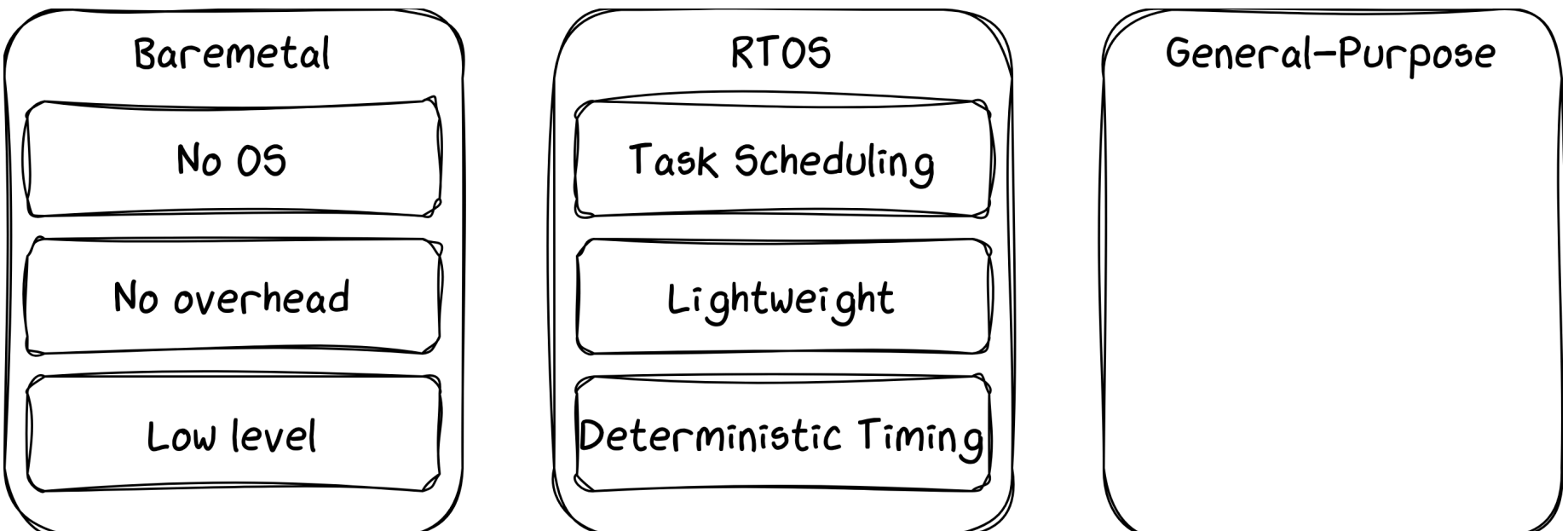
Embedded systems



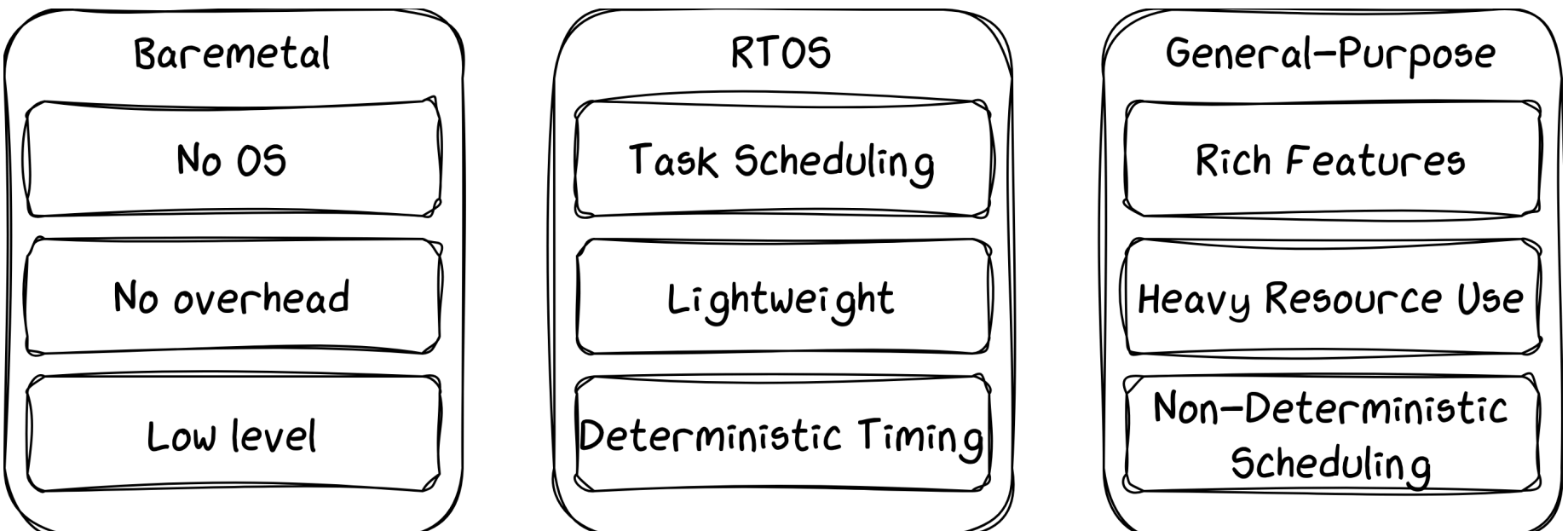
Embedded systems



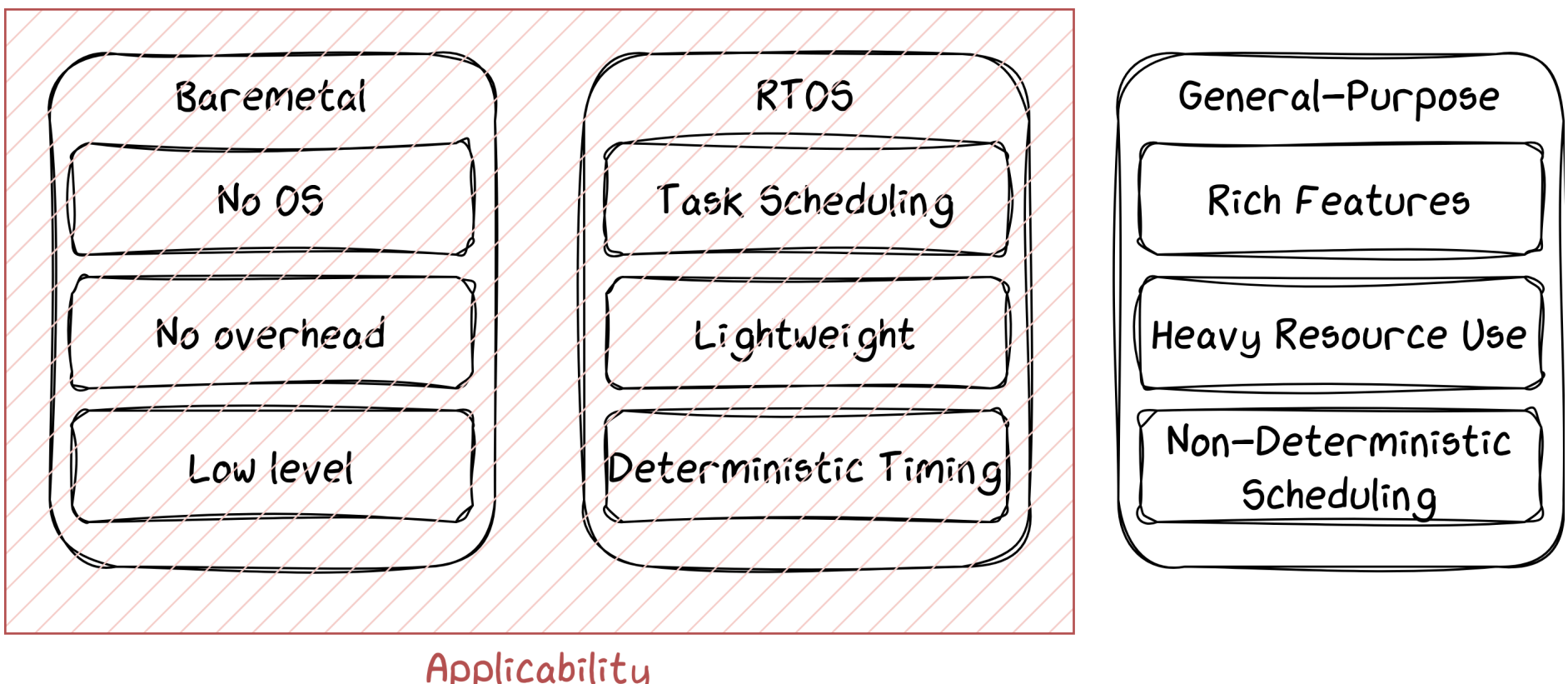
Embedded systems



Embedded systems



Embedded systems



A debugger?

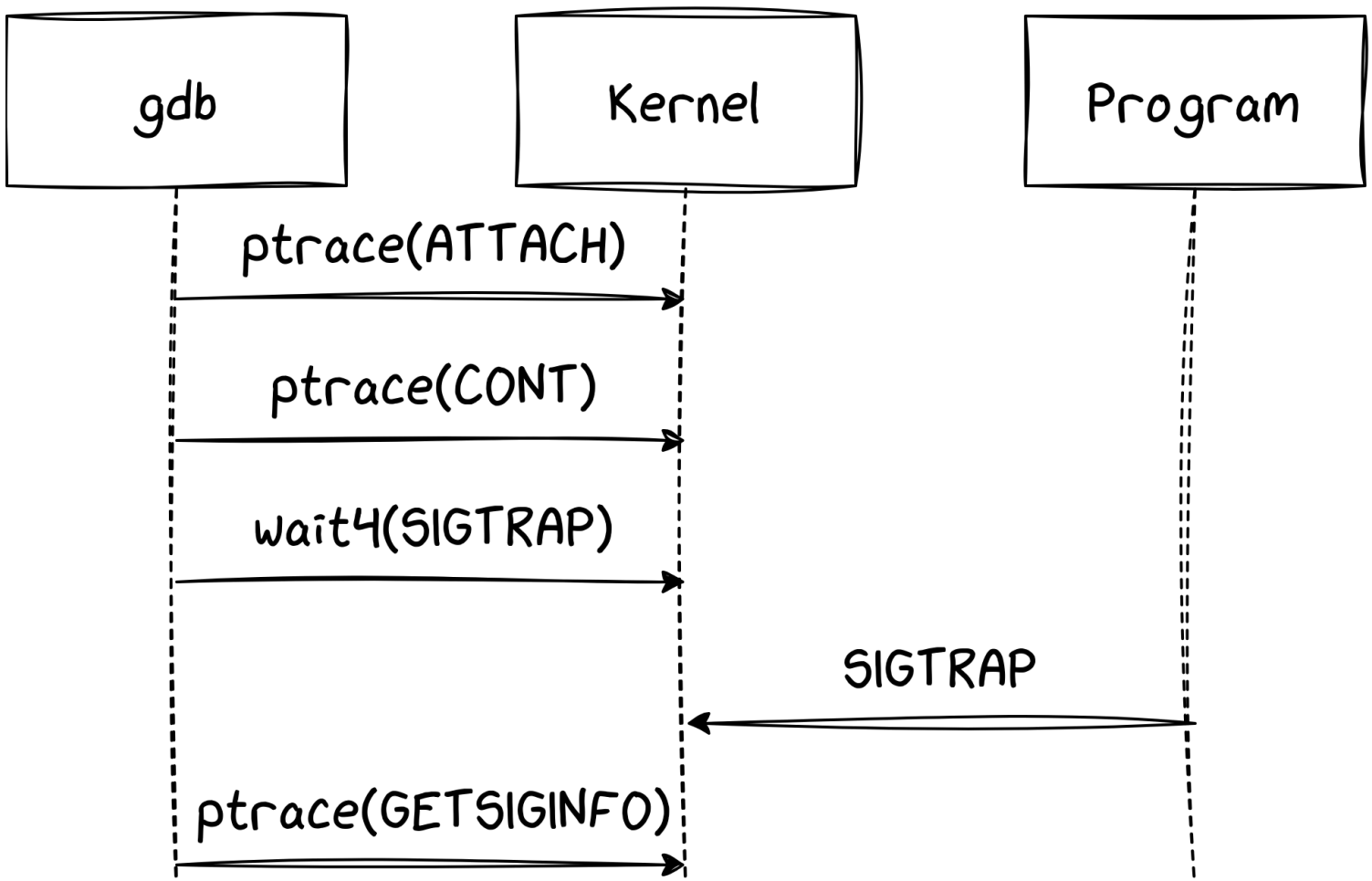
Basic features:

- Inspect memory and registers
- Set breakpoints

Useful for:

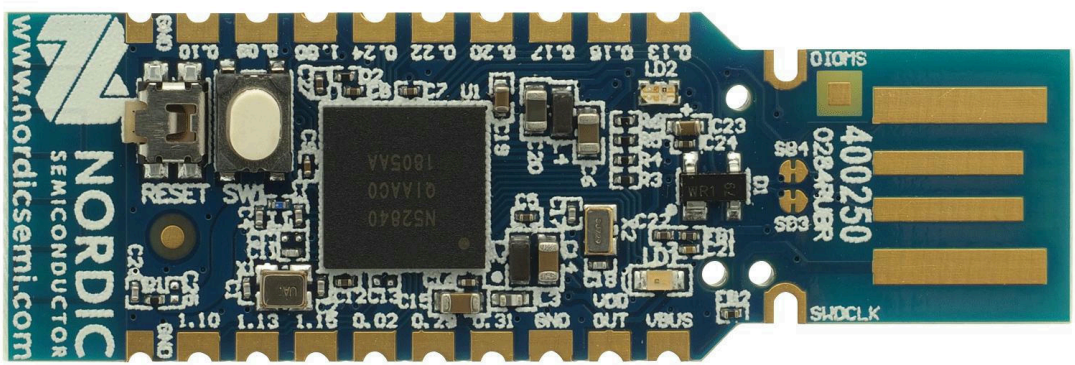
- Dynamic reverse-engineering
- Scripting
- Writing an exploit

A software debugger?



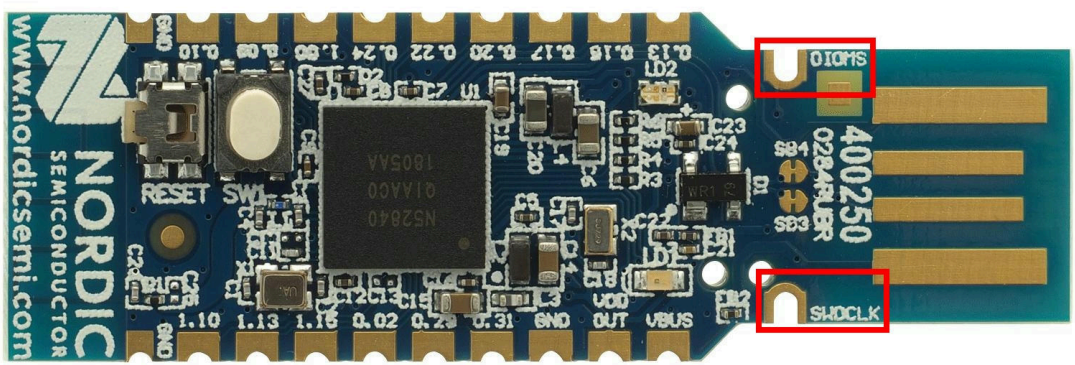
A hardware debugger?

- **Target hardware**: a few pins (JTAG, SWD or proprietary protocols)



A hardware debugger?

- **Target hardware**: a few pins (JTAG, SWD or proprietary protocols)



A hardware debugger?

- **Target hardware**: a few pins (JTAG, SWD or proprietary protocols)
- **Hardware**: ST-Link, JLink

A hardware debugger?

- **Target hardware**: a few pins (JTAG, SWD or proprietary protocols)
- **Hardware**: ST-Link, JLink
- **Interfacing Software**: OpenOCD, JLinkExe

A hardware debugger?

- **Target hardware**: a few pins (JTAG, SWD or proprietary protocols)
- **Hardware**: ST-Link, JLink
- **Interfacing Software**: OpenOCD, JLinkExe
- **Debugger**: gdb

But in production

Debug ports are disabled.

Thanks to: APPROTECT, RDP, etc.

So no hardware debugger

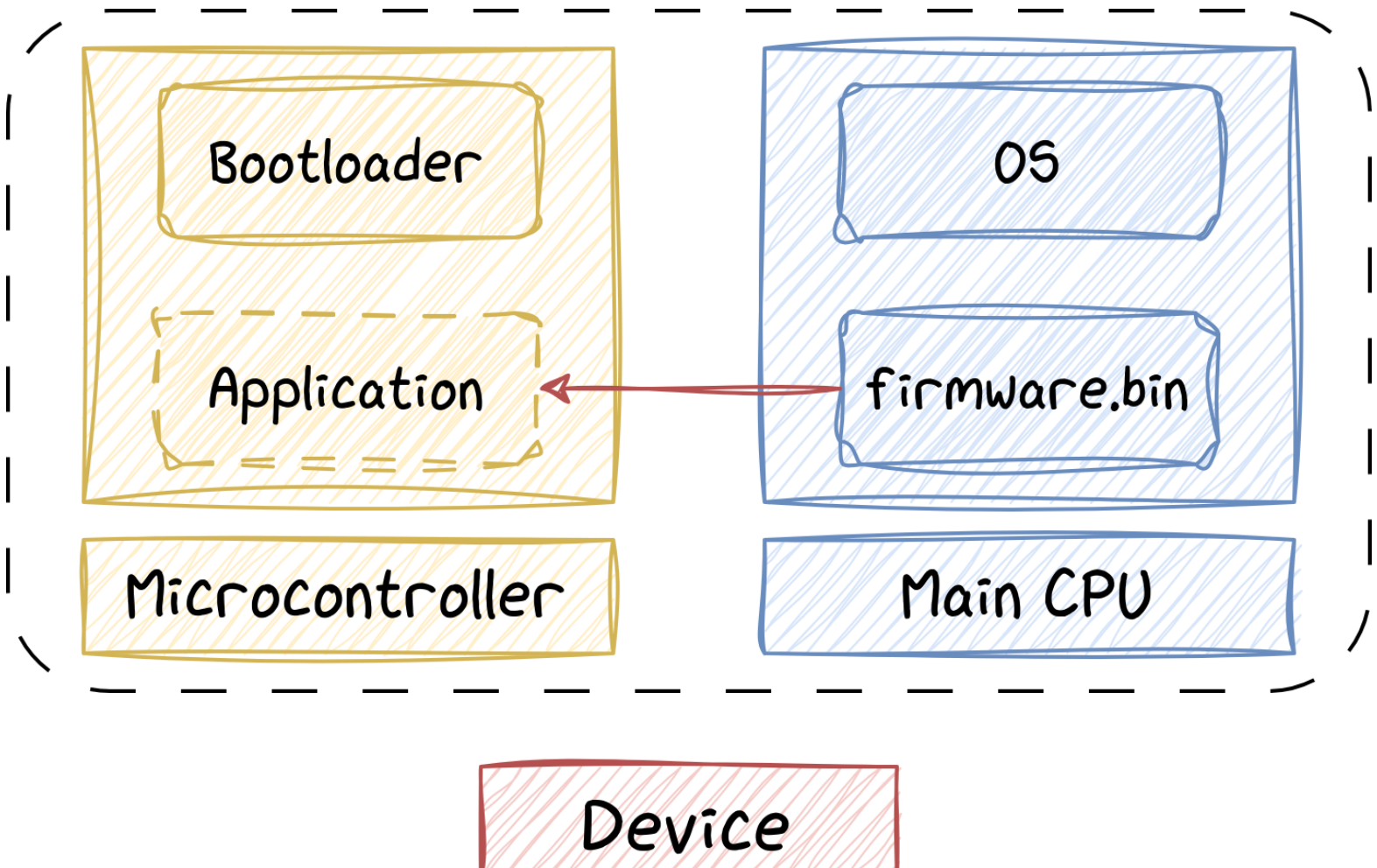
But in production

Debug ports are dis

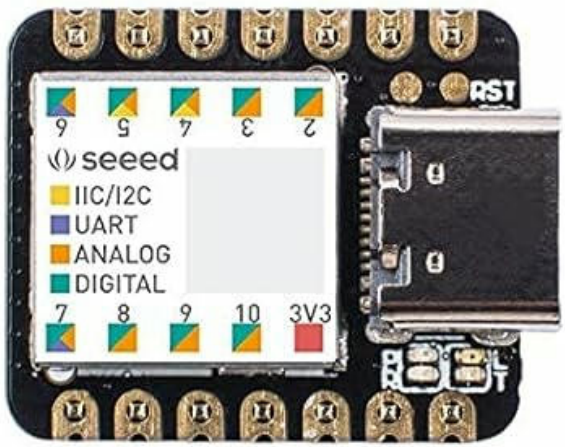
Thanks to: APPROT

So no hardware del

Sideloaded firmware



Context



ATSAMD21 - Cortex-M0+ - ARMv6-M

Context

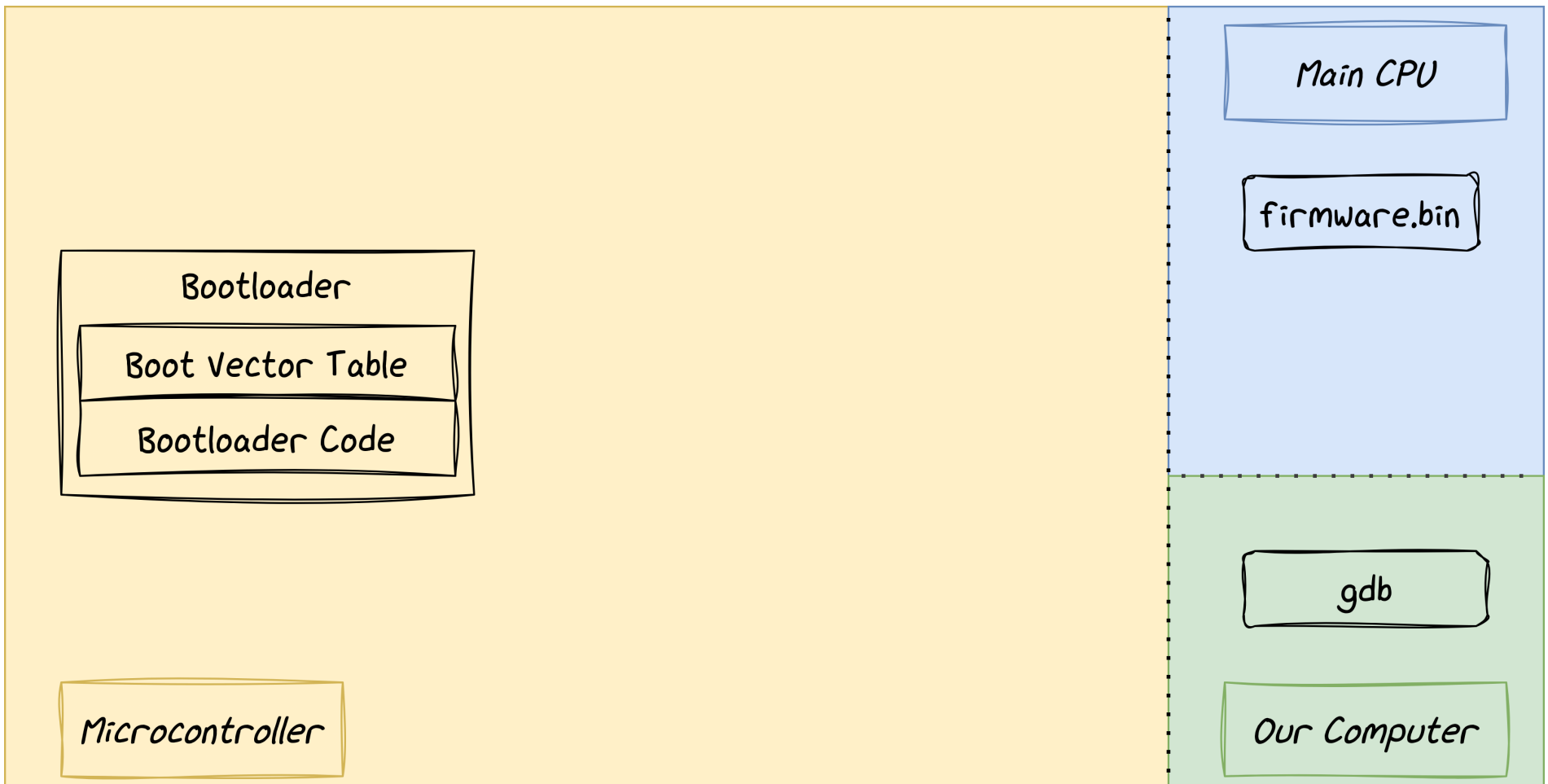
> ⚠ **Just one specific example.**
>
> What matters? The methodology

You need to **adapt** it to your:

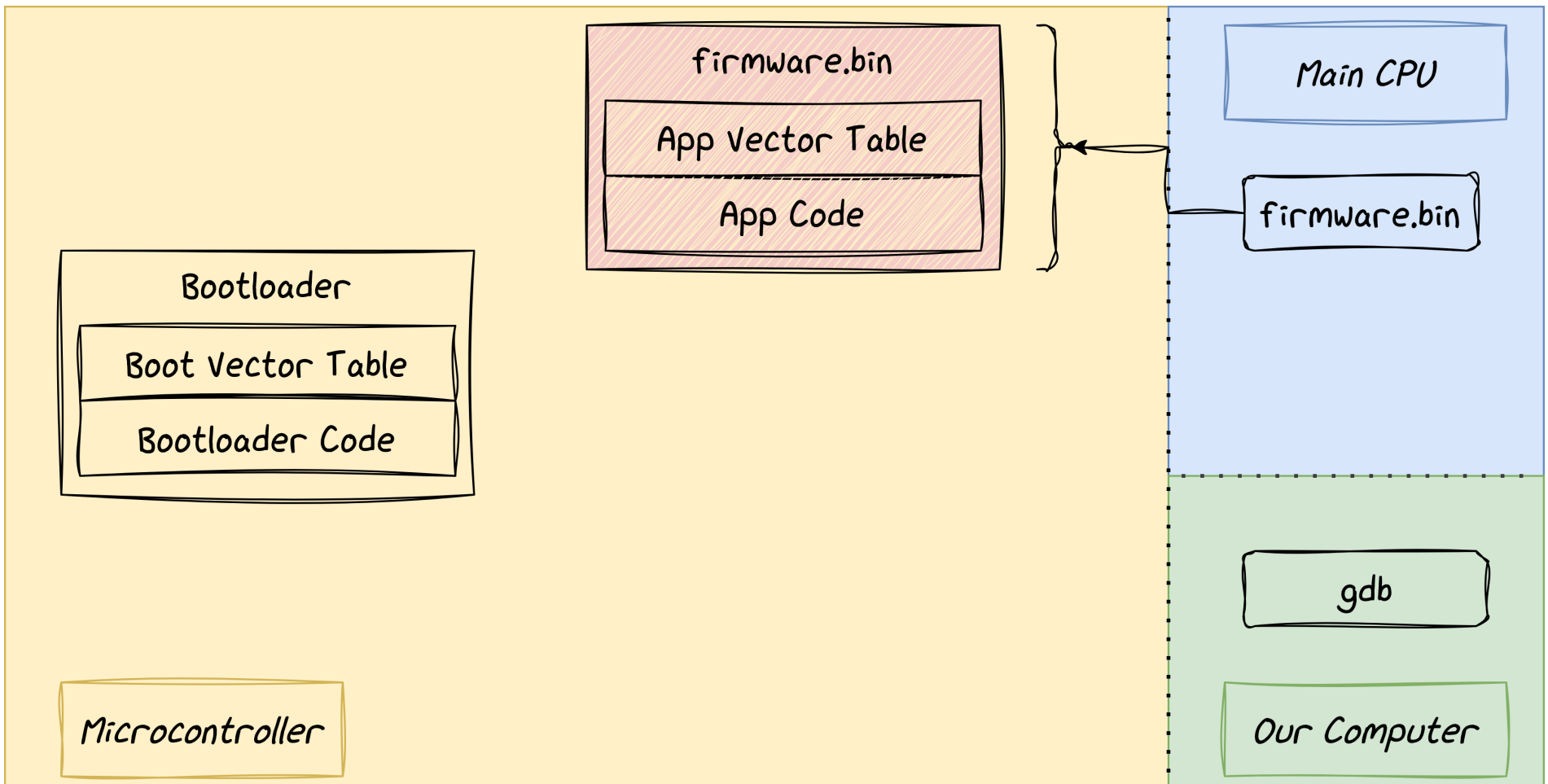
- architecture
- communication channel
- original firmware structure

Implementing the debugger

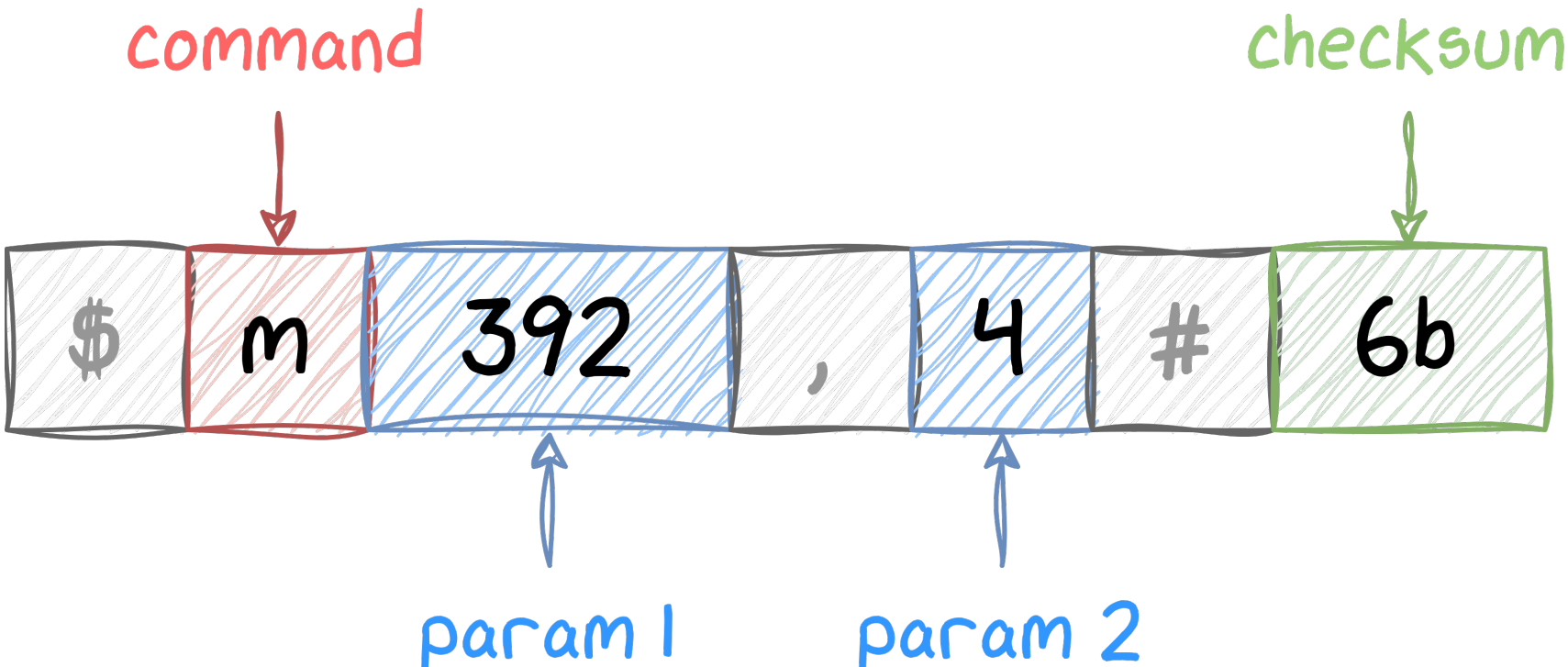
Another way


Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
gdb
Microcontroller
Our Computer

Another way


firmware.bin
App Vector Table
App Code
Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
gdb
Microcontroller
Our Computer

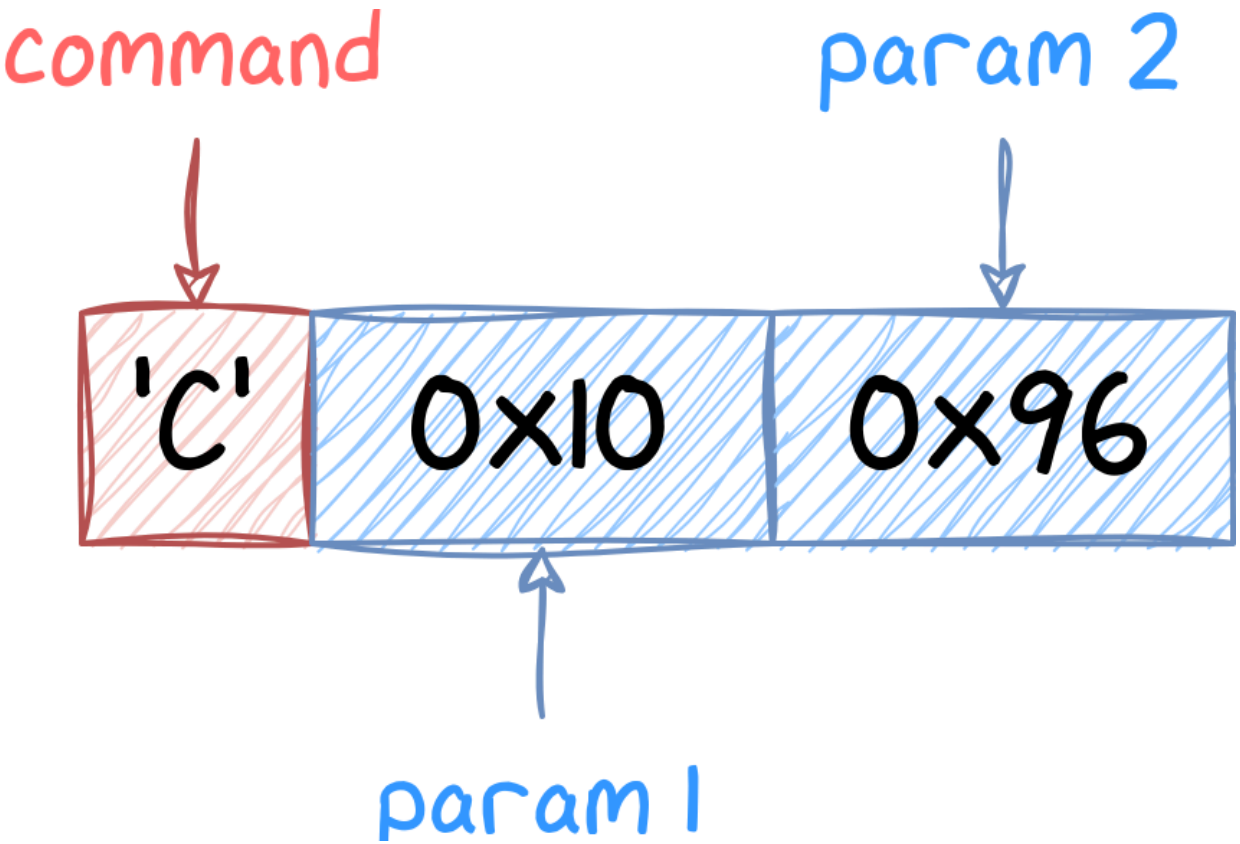
gdb remote serial protocol?



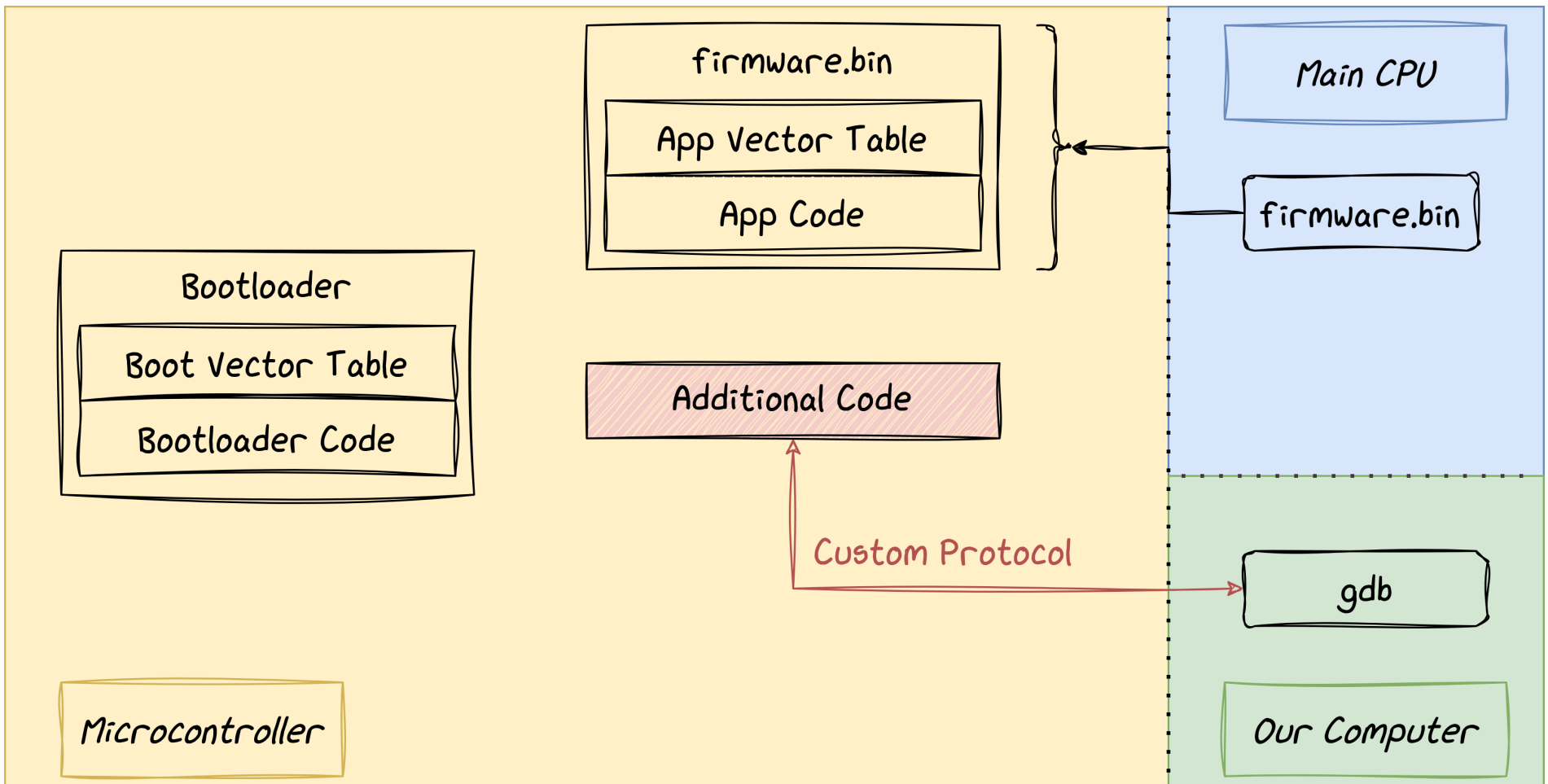
gdb protocol is too complex

- Query (`q`) and Set (`Q`) Packets
- `qSupported`
  - Negotiates optional features like:
    - `multiprocess+`
    - `no-resumed+`
    - `QStartNoAckMode`
    - `vContSupported+`
    - `xmlRegisters=i386` (or other target features)
- `qXfer:<object>:read` / `write`
  - Transfers structured data
  - Optional objects:
    - `features` – XML register layouts
    - `auxv` – Auxiliary vectors
    - `libraries` – Dynamic libraries loaded
    - `libraries-svr4`, `libraries-ld`
    - `memory-map` – Target memory regions (e.g. flash)
    - `threads` – Thread information
    - `exec-file` – Name of the running binary
- `qSymbol`
  - Dynamic symbol loading by GDB
- `qOffsets`
  - Communicates text/data segment relocation info (for PIE)
- `qAttached`
  - Indicates whether the debugger attached or created the process
- `qTStatus`, `qTf*`
  - Tracepoint support (rarely used)
- `QEnvironment:<VAR>=<VALUE>`
  - Set environment variables on target
- `QSetWorkingDir:<dir>`
  - Set working directory for target process
- `QStartNoAckMode`
  - Disables `+/-` acknowledgments to speed up communication
- `QPassSignals:<signal list>`
  - Configures which signals are passed to the inferior
- Execution Control
- `vCont`
  - Provides fine-grained control over thread execution (e.g., step/continue specific threads)
  - GDB sends `vContSupported` to discover available actions
- `vKill`
  - Terminates the target process
- `R`
  - Restart/soft reset the target
- `!`
  - Extended mode request (required for some stub use cases, but optional overall)
- Breakpoints & Watchpoints
- `Z` / `z` packets
  - Software/hardware breakpoints and watchpoints:
    - `Z0`, `z0`: Software breakpoint
    - `Z1`, `z1`: Hardware breakpoint
    - `Z2`, `z2`: Write watchpoint
    - `Z3`, `z3`: Read watchpoint
    - `Z4`, `z4`: Access watchpoint
  - Support is optional per type (e.g., a stub may support only software breakpoints)
- File and System I/O
- `F` / `0`
  - Host I/O via semi-hosting (deprecated, used in RedBoot-style stubs)
- `vFile:<op>`
  - Virtual file operations (e.g., open, read, write, stat, unlink, etc.)
  - Each operation is optional:
    - `vFile:open`
    - `vFile:read`
    - `vFile:close`
    - `vFile:unlink`
    - etc.
- Thread & Multiprocessing Support
- `Hc` / `Hg`
  - Set current thread for continue or register access
- `qfThreadInfo` / `qsThreadInfo`
  - Enumerate threads
- `qThreadExtraInfo`
  - Provide additional info (like thread names)
- `qC`
  - Report current thread
- `multiprocess+`
  - Declared via `qSupported` if stub supports multiple processes

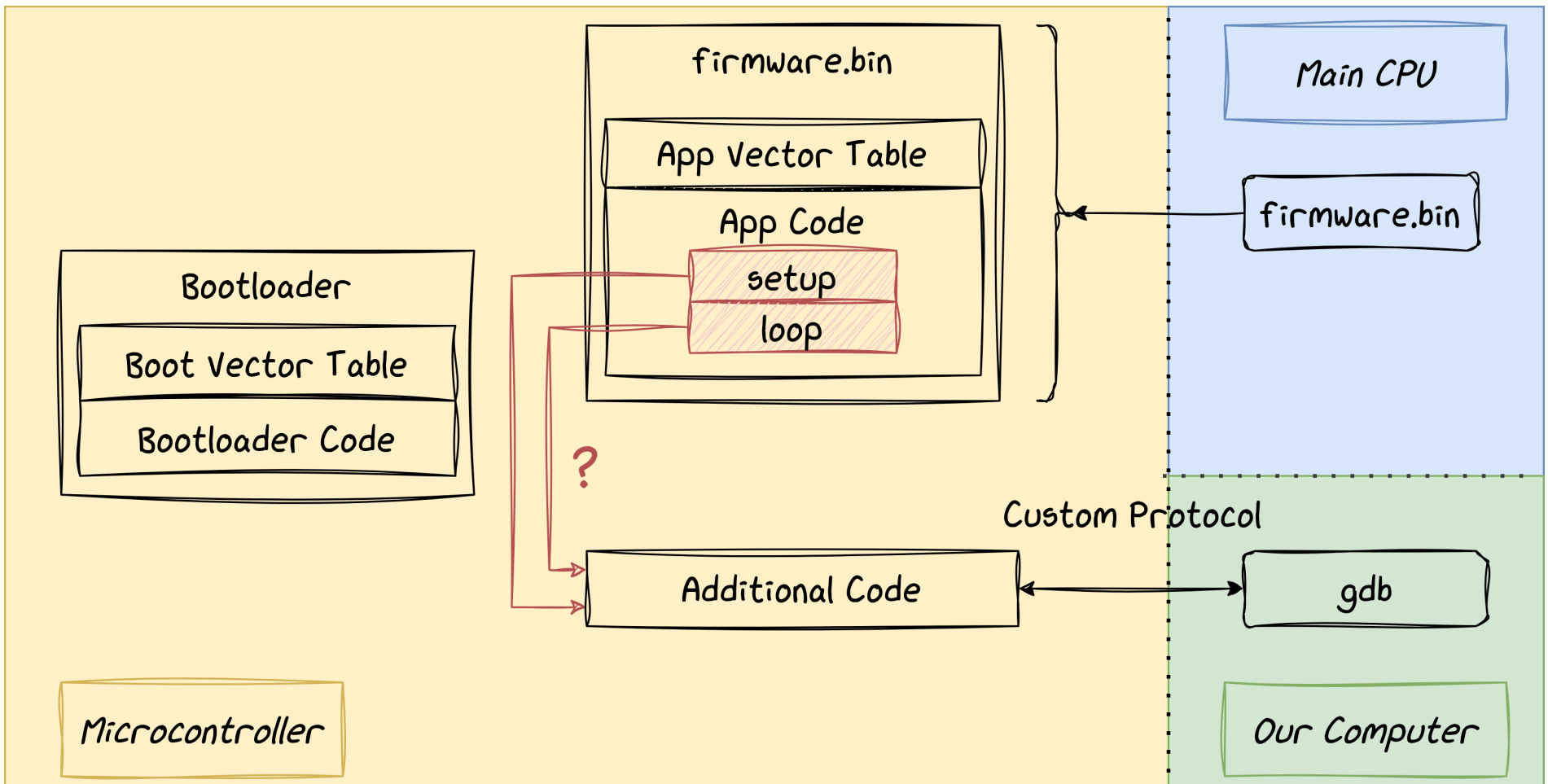
A simpler debug protocol



Next step


firmware.bin
App Vector Table
App Code
Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
Additional Code
Custom Protocol
gdb
Microcontroller
Our Computer

Next step


firmware.bin
App Vector Table
App Code
setup
loop
Bootloader
Boot Vector Table
Bootloader Code
?
Additional Code
Main CPU
firmware.bin
Custom Protocol
gdb
Microcontroller
Our Computer

Initial call to the debugger

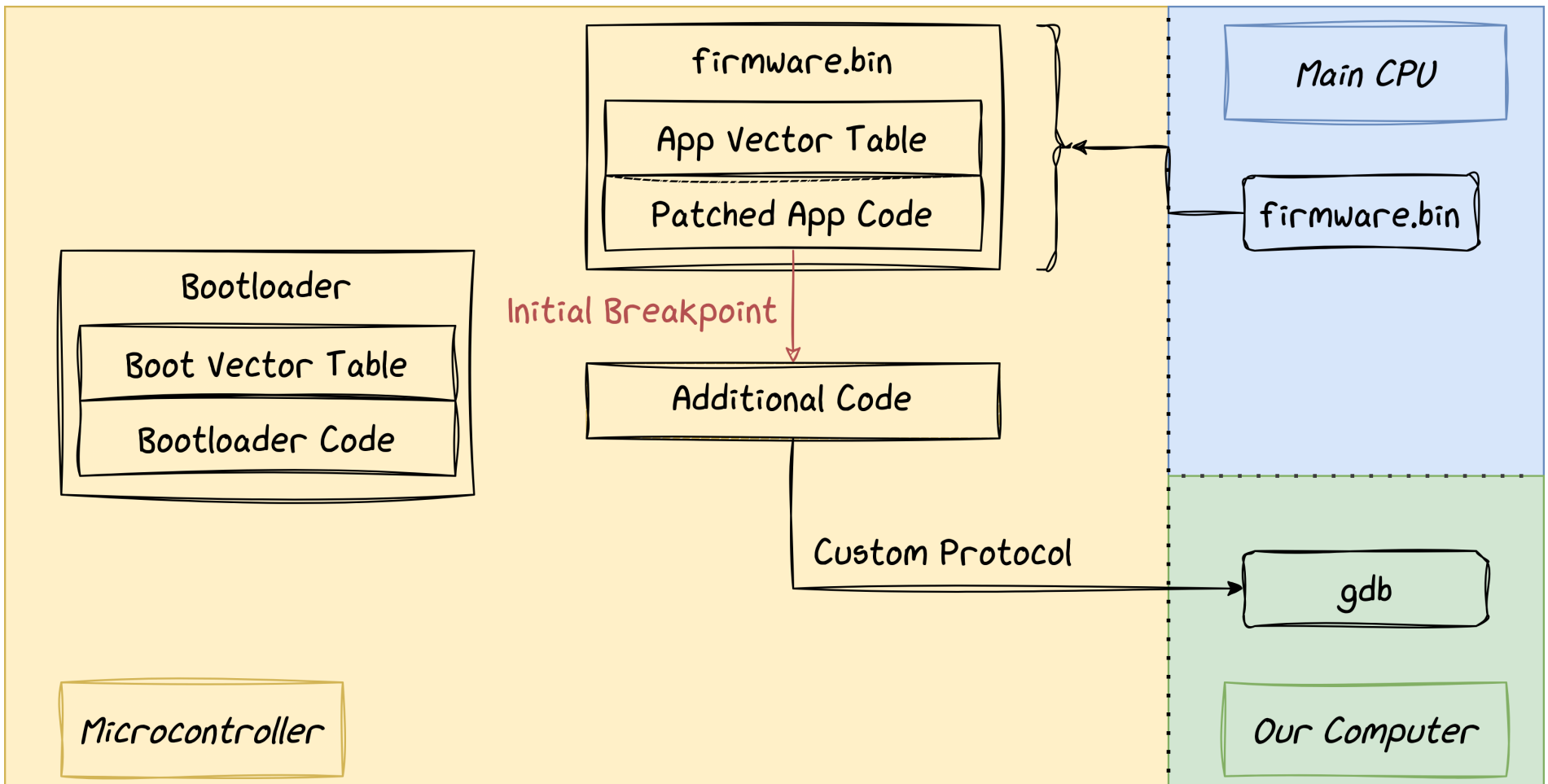
- **`setup`**
  - can debug the `setup` function
  - have to wait for communication channel initialization

Initial call to the debugger

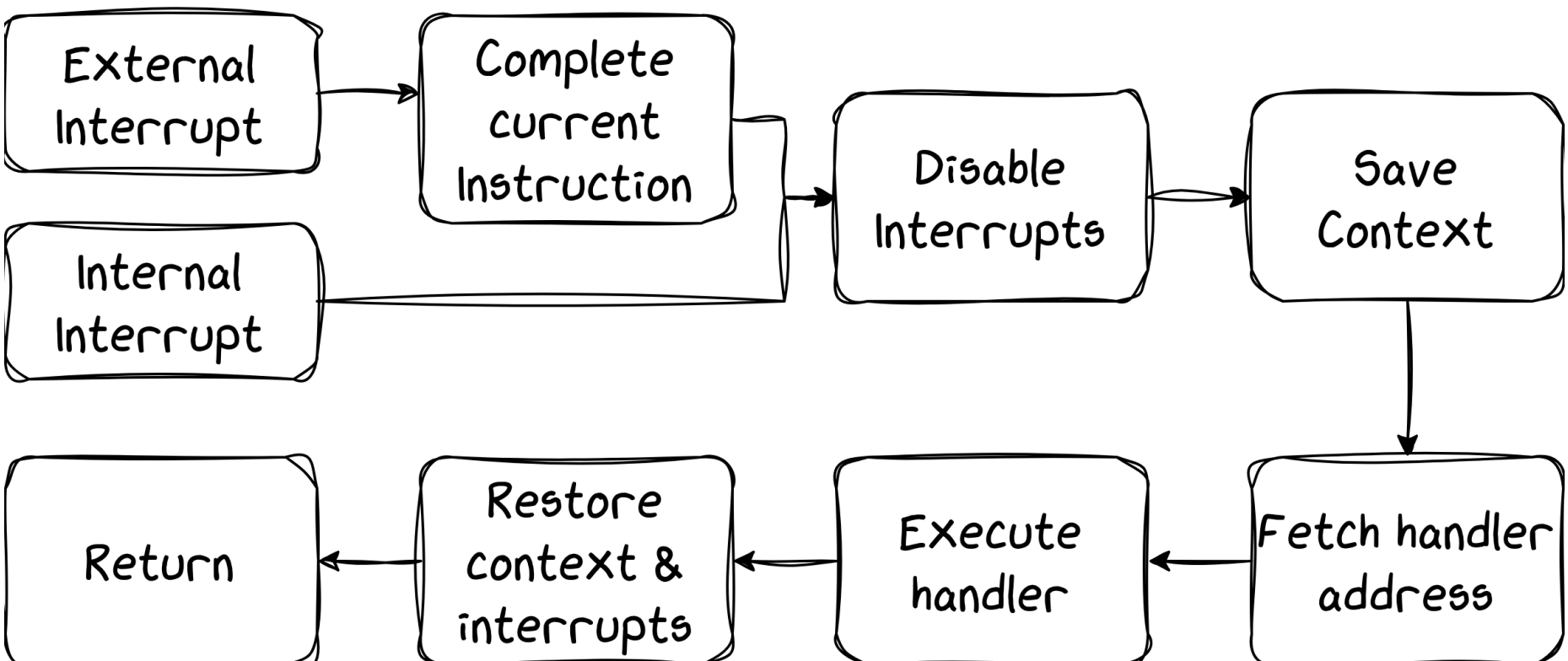
- **`setup`**
  - can debug the `setup` function
  - have to wait for communication channel initialization

- **`loop`**
  - can check if user pressed `Ctrl+C` in `gdb`
  - cannot debug the `setup` function
  - need to handle this breakpoint differently

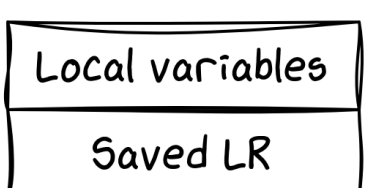
Next step


firmware.bin
App Vector Table
Patched App Code
Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
Initial Breakpoint
Additional Code
Custom Protocol
gdb
Microcontroller
Our Computer

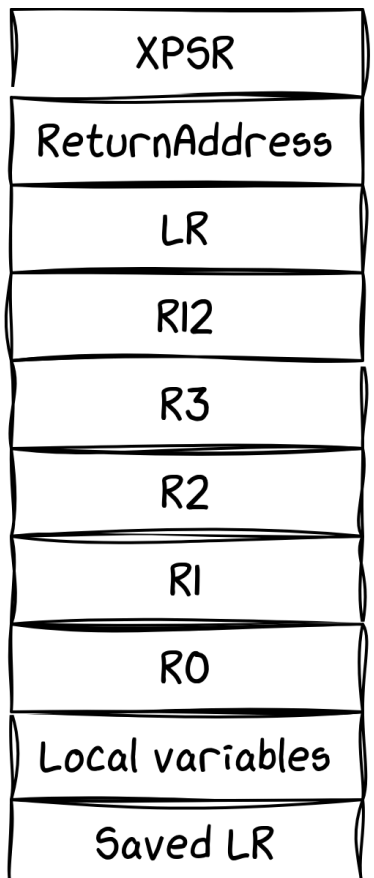
Interrupt flow



Registers saving in ARMv6-M

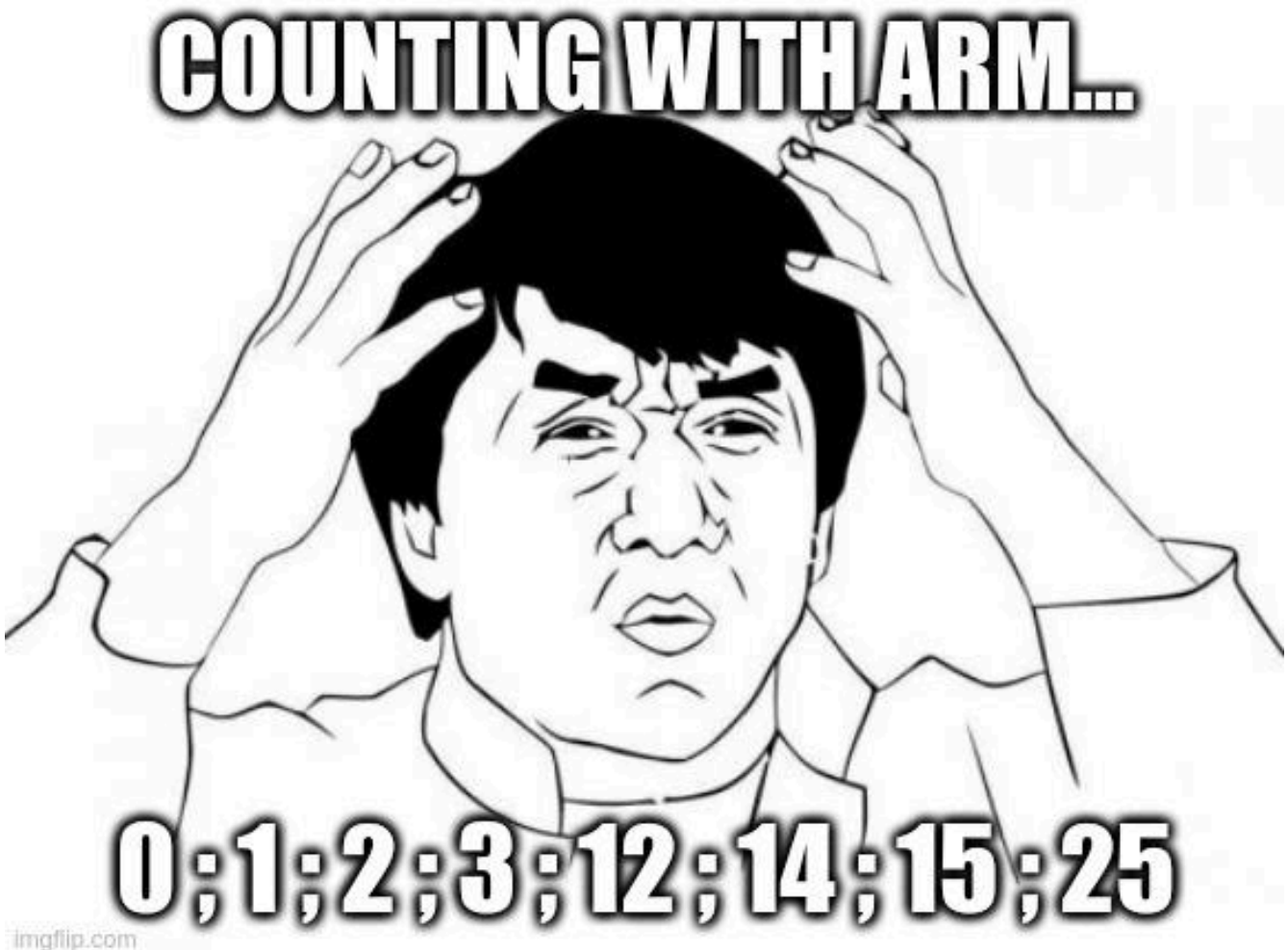


Before an interrupt

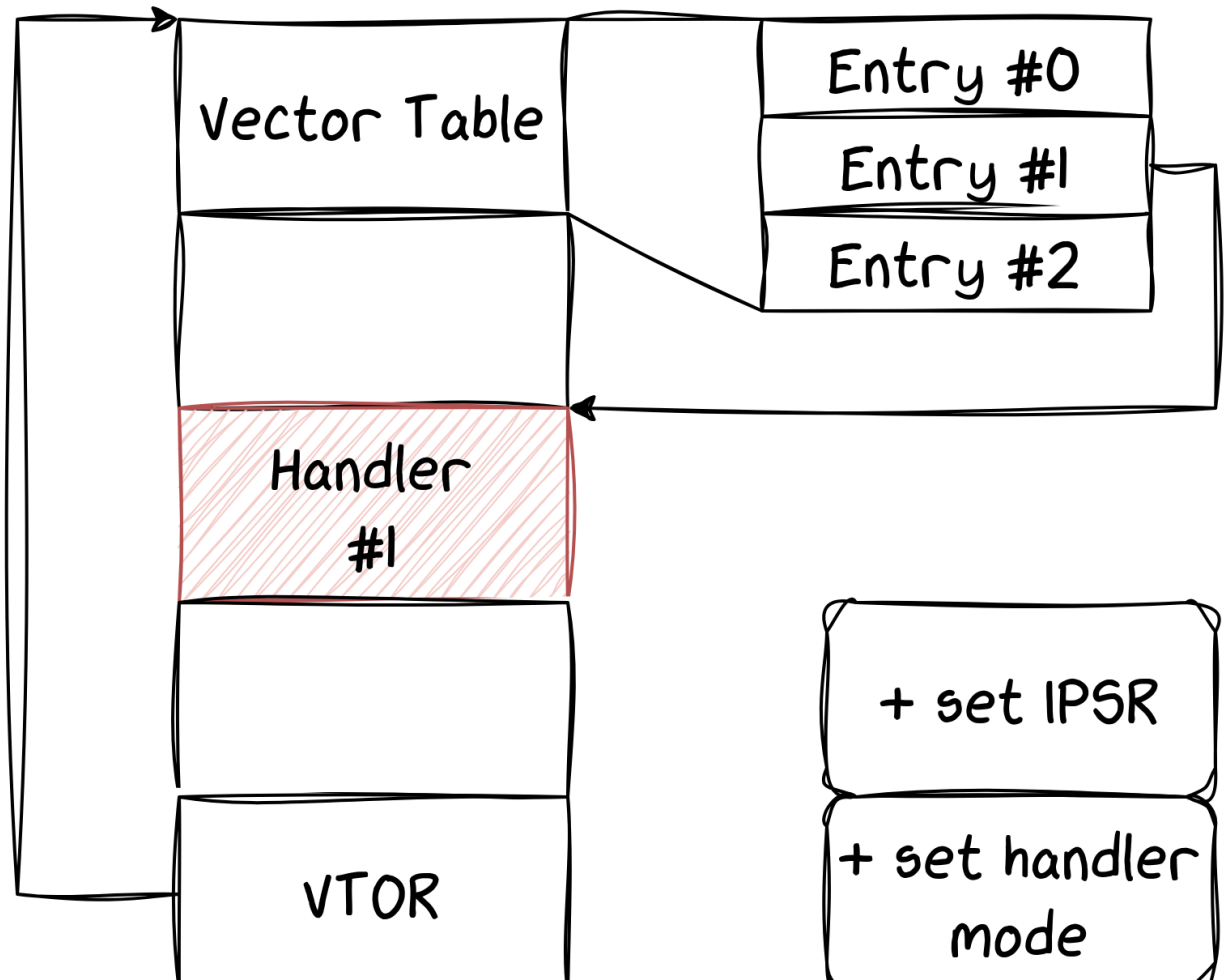


After an interrupt

Registers saving in ARMv6-M



Exception entry behaviour in ARMv6-M



Vector table in ARMv6-M

Exception #	Vector Name	Description
-	Initial SP Value	Initial Stack Pointer (top of stack)
1	Reset	Reset handler
2	NMI	Non-Maskable Interrupt
3	HardFault	Hard Fault handler
4-10	Reserved	Reserved for future use
11	SVCall	Supervisor Call handler
12-13	Reserved	Reserved for future use
14	PendSV	Software-generated Supervisor Calls
15	SysTick, optional	SysTick timer handler
16	External Interrupt(0)	External interrupt handler 0
16 + N	External Interrupt(N)	External interrupt handler N

Vector table in ARMv7-M

Exception #	Vector Name	Description
-	Initial SP Value	Initial Stack Pointer (top of stack)
1	Reset	Reset handler
2	NMI	Non-Maskable Interrupt
3	HardFault	Hard Fault handler
4	MemManage	-
5	BusFault	-
6	UsageFault	-
7-10	Reserved	Reserved for future use
11	SVCall	Supervisor Call handler
12	**DebugMonitor**	Called when encountering a breakpoint
...	...	...

Implementing the debug handler

In assembly for:

- direct access to registers
- avoiding dangerous optimizations
- using tricks to make the code shorter

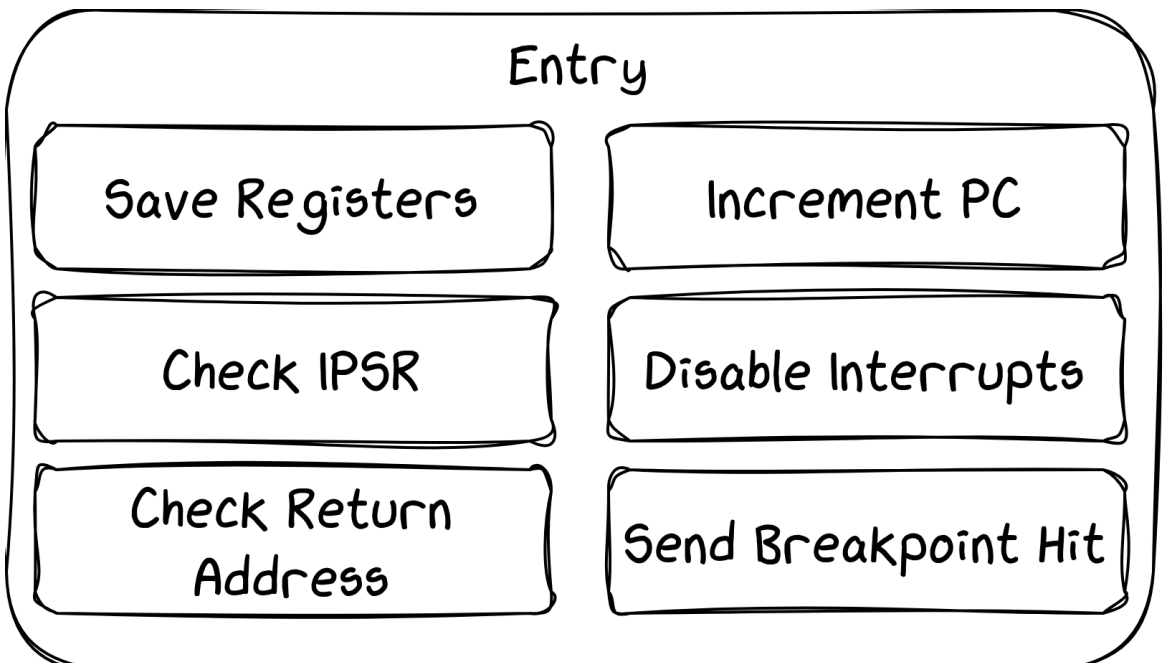
Implementing the debug handler

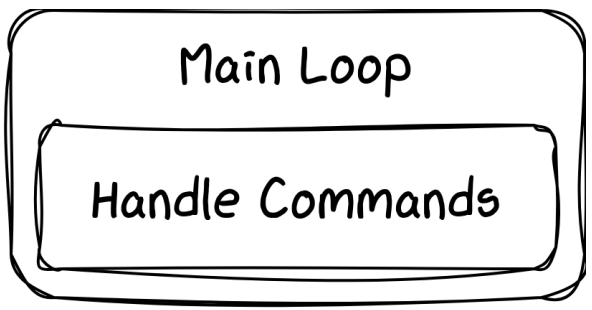
Implementing the debug handler

Implementing the debug handler

Implementing the debug handler

Implementing the debug handler





Reading a register: the boring way?

```
uint32_t read_register(int reg_number);
```

```
read_register:
  CMP  R0, #0
  BEQ  read_R0
  CMP  R0, #1
  BEQ  read_R1
  CMP  R0, #2
  BEQ  read_R2
  CMP  R0, #3
  BEQ  read_R3
[...]
```

```
read_R0:
  MOV  R0, R0
  BX   LR
read_R1:
  MOV  R0, R1
  BX   LR
read_R2:
  MOV  R0, R2
  BX   LR
[...]
```

Reading a register: the boring way?

```c
uint32_t read_register(int reg_number);
```

```
read_register:
  LSLS R0, R0, #2
  ADD  R0, pc
  ADDS R0, #4
  BX   pc
[...]
```

```
read_R0:
  MOV  R0, R0
  BX   LR
read_R1:
  MOV  R0, R1
  BX   LR
read_R2:
  MOV  R0, R2
  BX   LR
[...]
```

Reading a register: jump table

```
read_register:
    PUSH {R0-R15}
    LSLS R1, R0, #2
    ADD  R1, SP, R1
    LDR  R0, [R1]
    BX   LR
```

> ⚠ **Problem**
>
> No `PUSH {R0-R15}` in ARMv6-M (only `PUSH {R0-R7}`).

Cool trick

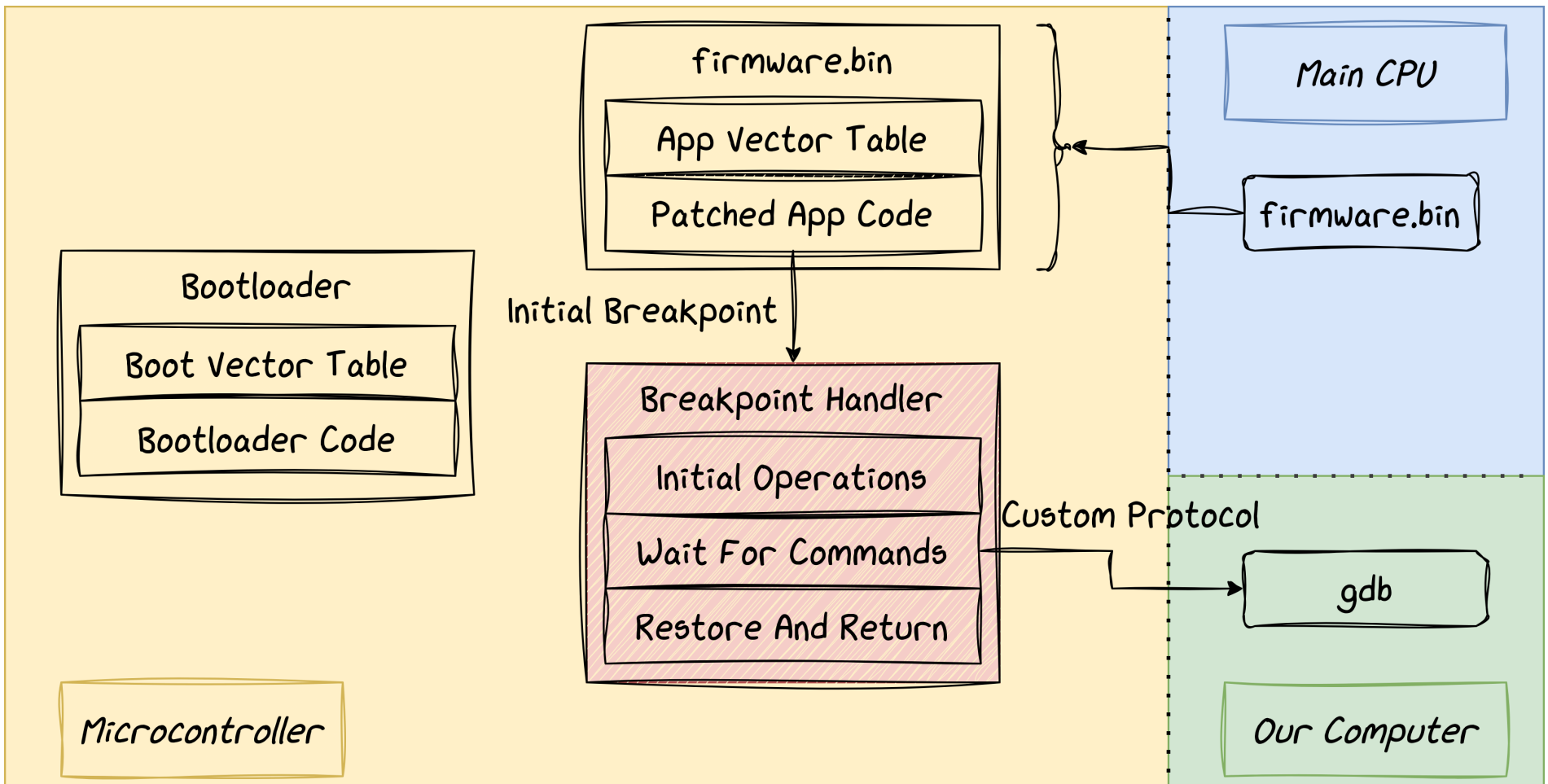
```
    ldr R1, =code_block
    ldr R2, =0xE1A00000       ; Base opcode for NOP (MOV R0, R0)
    add R2, R2, R0, lsl #12   ; Modify opcode to MOV R0, Rn
    str R2, [R1]

    dsb                       ; Data Synchronization Barrier
    isb                       ; Instruction Synchronization Barrier

code_block:
    nop                       ; Instruction to modify
    bx  lr
```

Useful for system registers

It can be done with:

- MRS (Move to Register from Special register)
- MSR (Move to Special Register from ARM Register)

Next step


firmware.bin
App Vector Table
Patched App Code
Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
Initial Breakpoint
Breakpoint Handler
Initial Operations
Wait For Commands
Restore And Return
Custom Protocol
gdb
Microcontroller
Our Computer

Write a gdb server?

- gdbserver
- openocd
- qemu

Write a gdb server?

- gdbserver

2,095,581 lines of C

- openocd

6,178 lines of C

- qemu

3,701 lines of C

Write a gdb server?

- gdbserver

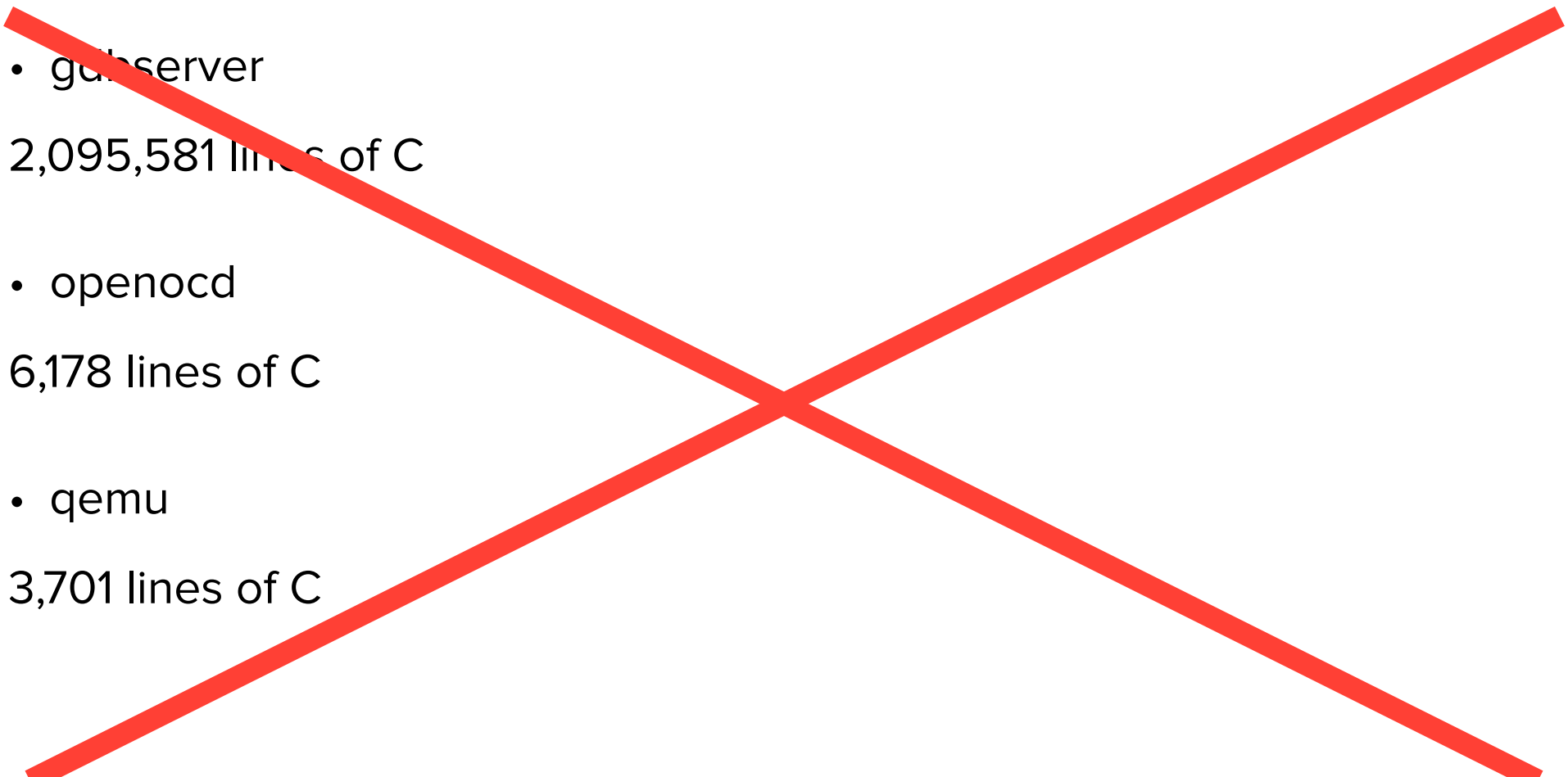
2,095,581 lines of C

- openocd

6,178 lines of C

- qemu

3,701 lines of C



Write a simple gdb server?

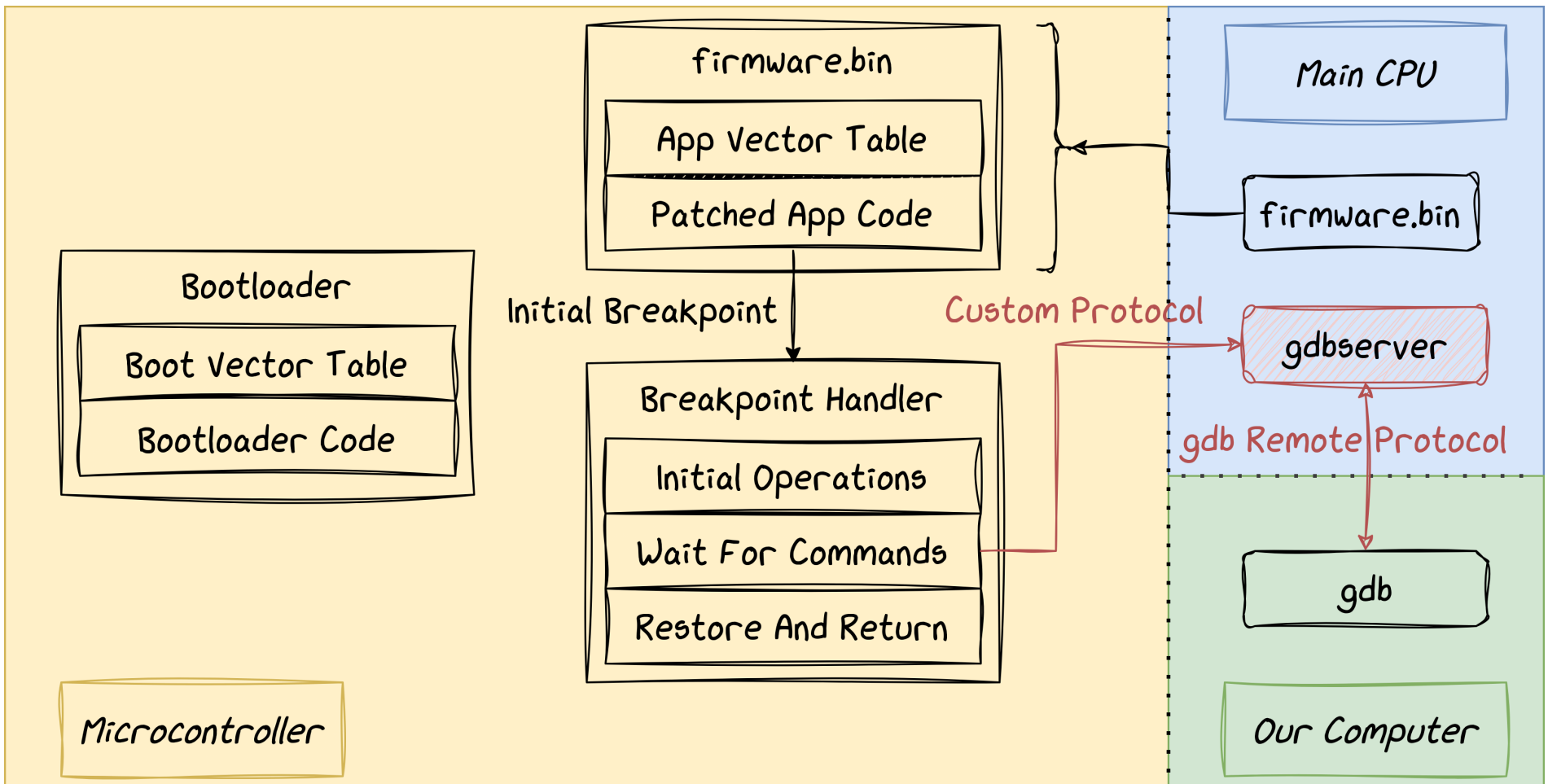
Simple open source project `minigdbstub`[1]

Just need to implement:

```
void minigdbstubUsr::Putchar(char data);
char minigdbstubUsr::Getchar();
void minigdbstubUsr::WriteMem(size_t addr, unsigned char data);
unsigned char minigdbstubUsr::ReadMem(size_t addr);
void minigdbstubUsr::Continue();
void minigdbstubUsr::Step();
void minigdbstubUsr::ProcessBreakpoint(int type, size_t addr);
void minigdbstubUsr::KillSession();
```

[1] https://github.com/annestrand/minigdbstub

Next step


firmware.bin
App Vector Table
Patched App Code
Main CPU
firmware.bin
Bootloader
Boot Vector Table
Bootloader Code
Initial Breakpoint
Custom Protocol
gdbserver
Breakpoint Handler
Initial Operations
Wait For Commands
Restore And Return
gdb Remote Protocol
gdb
Microcontroller
Our Computer

Demonstration

Conclusion

No hardware debugger but firmware updatable?

Debugging is still **possible** and **not so hard**!

Conclusion

No hardware debugger but firmware updatable?

Debugging is still **possible** and **not so hard**!

Special thanks to virtualabs and MadSquirrels.

Thank you

Contact information

Email:	contact@quarkslab.com
Phone:	+33 1 58 30 81 51
Website:	https://www.quarkslab.com

Quarkslab