Lecture 06
Timers

CE346 – Microcontroller System Design

Branden Ghena – Fall 2024

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Administrivia

- Last chance for Lab1 checkoffs 5:00-6:00 today
  - Quite a few groups remaining, so it'll likely focus on checkoffs first
- Don't forget to answer the postlab questions on Gradescope
  - You and your partner can work on them together, but submit separately
- Lab2 tomorrow! Virtual Timer Lab
- Project proposals due next week Thursday!
  - Be sure to find a group. Fill out the survey if you want to find someone!

Next week Tuesday: online recording

- Unfortunately, I'll be out-of-town on Tuesday next week
- So, no in-person class on that day
- I will record the lecture in advance and put it out on Panopto
    - Lecture on Sensors
    - Moved the schedule around a little

Today’s Goals

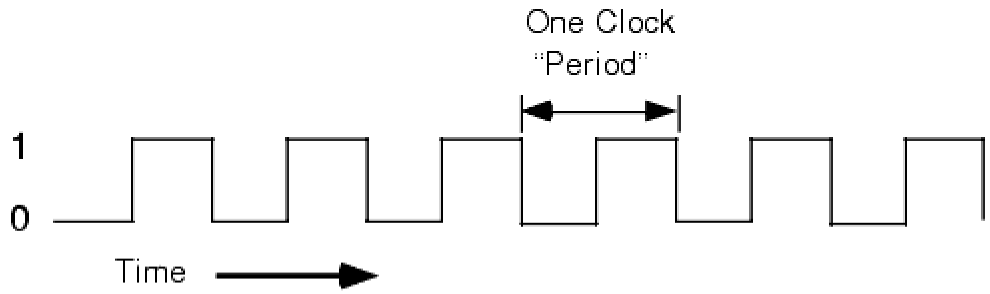
- Understand the role of clocks in a microcontroller
- Explore functionality of various timer peripherals on the Microbit

Outline

- **Clocks**
- Timers
- Virtualizing Resources
- Real-Time Counter
- Watchdog

What are clocks?

- Clock signals, in the microcontroller context, are oscillating square wave signals used to switch transistors and latch inputs

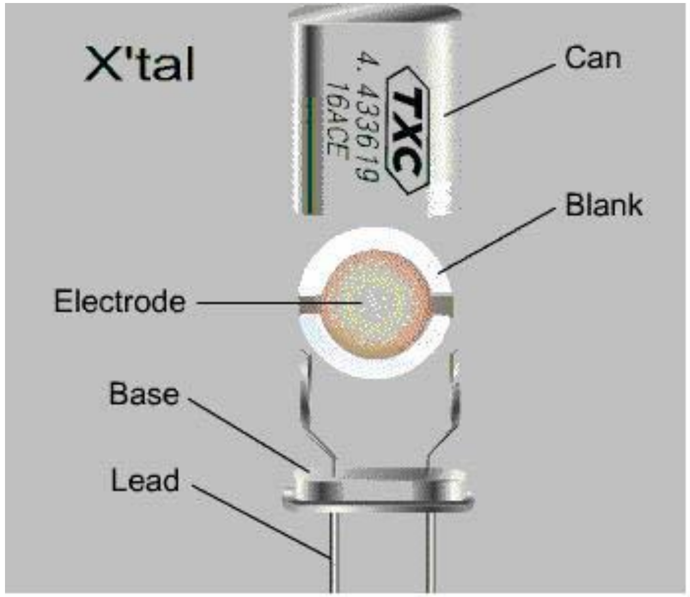


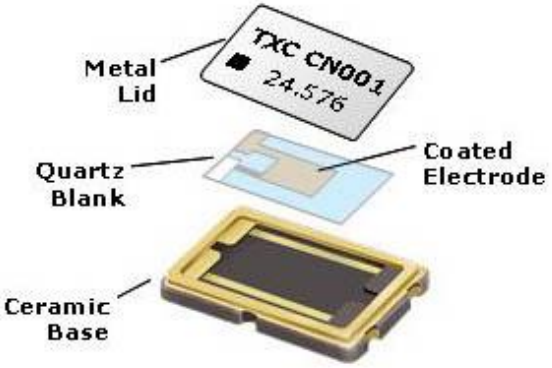
- A clock MUST be running for (almost) anything on a microcontroller to function (processor and peripherals)
  - Exceptions:
    - Low-power input interrupts
      - GPIOTE port interrupt, Analog LPCOMP interrupt, NFC sense interrupt, USB power interrupt
    - Reset signal

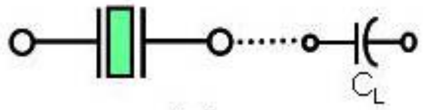
Generating clocks

- External crystal oscillator
  - Creates clock signal
  - Chunk of quartz
  - Behaves like RLC circuit but uses less energy

- Internal mechanisms
  - RC oscillator
    - Creates clock signal
    - Less accurate and higher energy than crystal
  - Phase-Locked Loop (PLL)
    - Multiply input to create new higher frequency clocks

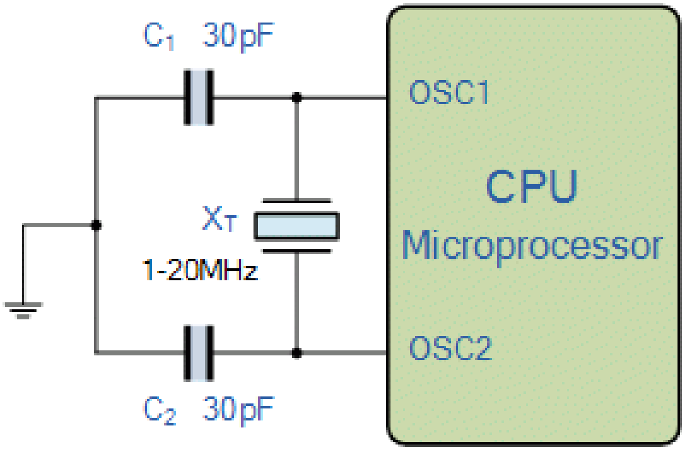


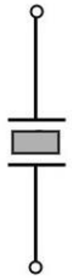




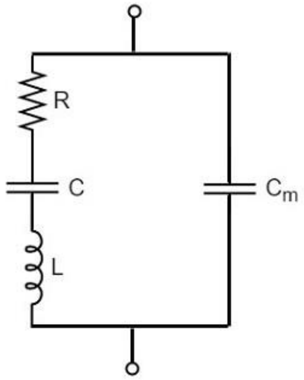
(c)

(Fig.7) (a) Metal can type resonator
(b) Ceramic SMD type resonator
(c) Symbol of crystal unit





Symbol of a Crystal

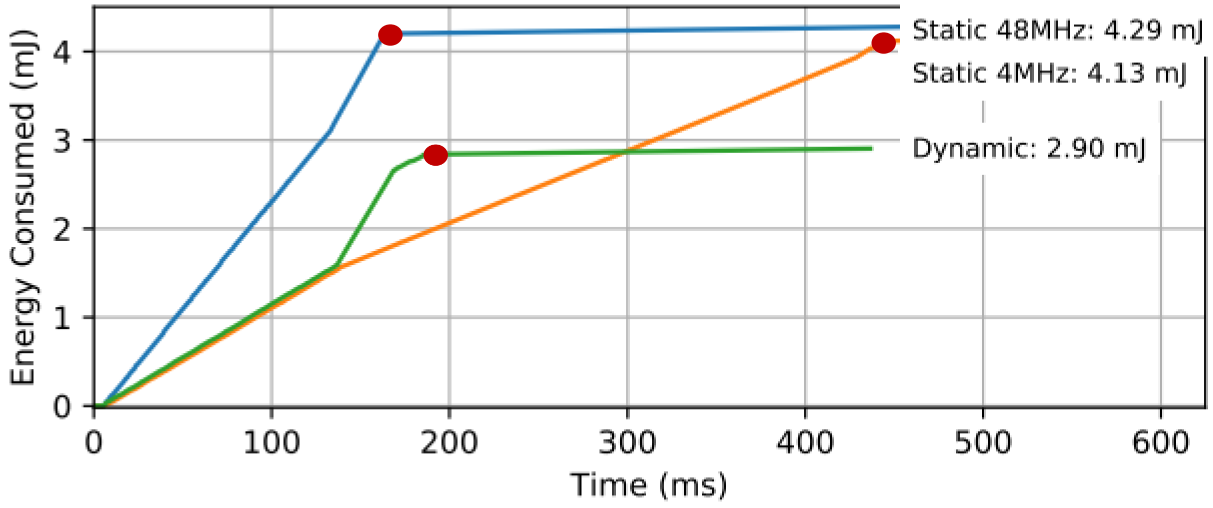


Equivalent circuit of a crystal

Microbit crystal for nRF52833

Clocks and energy

- Fundamental tradeoff
  - Faster clock gets things done faster but uses more energy
  - Slower clock uses less energy but gets things done slower
  - Which to use depends on the situation
    - CPU bound: faster clock, IO bound: slower clock



Example of clock selection for a mixed load (part IO, part CPU)

Energy consumed becomes a horizontal line when the task is completed

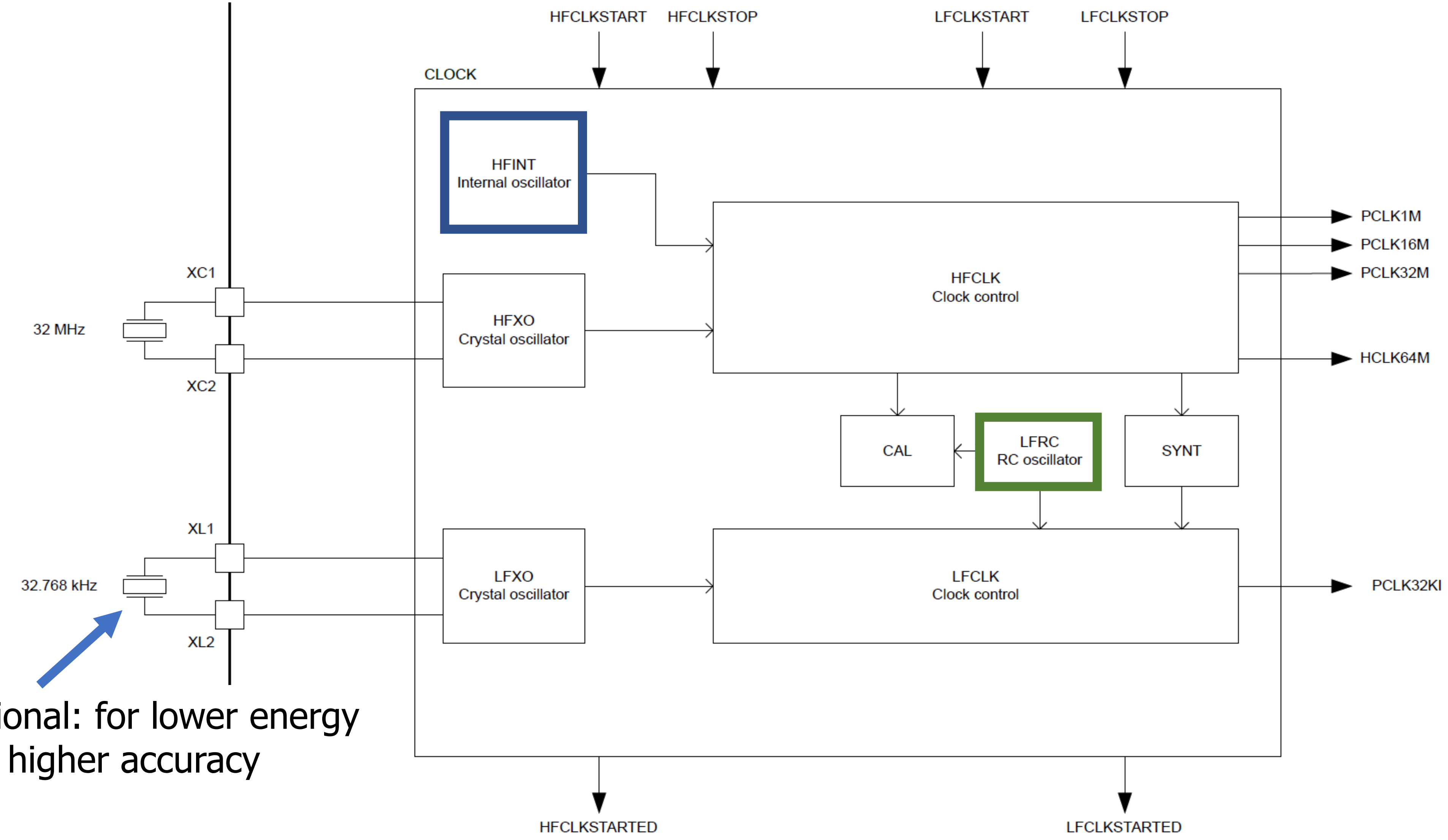
Chiang et al. "[Power Clocks: Dynamic Multi-Clock Management for Embedded Systems]" EWSN 2021

Controlling clocks

- Some microcontrollers provide extremely fine-grained control over clocks
  - Really complicated section of code to get working
  - Many combinations are invalid
  - Manually enable/disable clocks as needed

- nRF52 instead gives almost no control but is easier to use
  - One 64-MHz clock for processor
  - Multiple peripheral clocks, but (most) peripherals are hardwired to one
    - 16 MHz for almost all peripherals (PDM and I2S are 32 MHz)
  - Low-frequency 32 kHz clock for low-power peripherals
  - Automatically enables/disables clocks

nRF52833 clocks



Optional: for lower energy and higher accuracy

Electrical characteristics

- Active power of clocks
  - 32 kHz crystal run current: 0.23 μA
  - 32 kHz RC oscillator run current: 0.70 μA
  - 32 MHz crystal average run current: 300-700.00 μA
  - 32 MHz standby current: 110.00 μA

- Startup time for external crystals
  - 32 kHz crystal: 250-500 ms (milliseconds!!!)
  - 32 MHz crystal: 60-200 μs
  - Beware: switching can lead to delays and instability
    - nRF52 uses RC oscillator while crystal is not yet ready

Outline

- Clocks
- **Timers**
- Virtualizing Resources
- Real-Time Counter
- Watchdog

Timer peripherals

- Common need for embedded systems: sense of time
  - Start this behavior after a certain amount of time
  - Stop this behavior after a certain amount of time
  - Measure how much time passed between two events

- Timer peripherals
  - Input is one of the system clocks
  - Counts up a register at each clock tick
    - Looking at register at start and end can give real-world duration
  - Compare to saved value and trigger interrupt on match
    - Allows interrupts to be scheduled in the future

Break + Discussion

- What is the finest granularity you might need from a timer?
  - Give an example of the use case

- What is the longest duration you might need from a timer?
  - Give an example of the use case

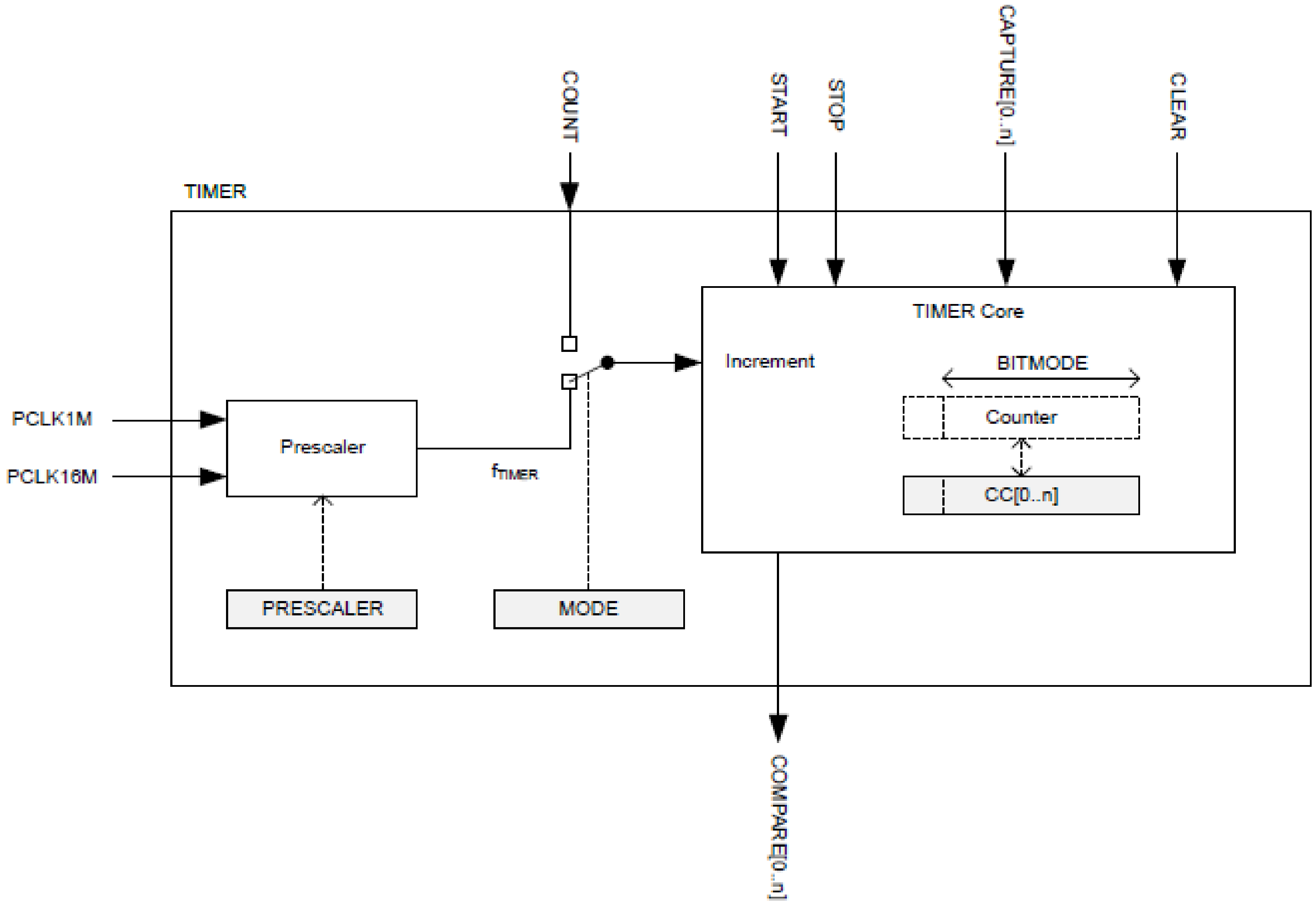
Break + Discussion

- What is the finest granularity you might need from a timer?
  - Give an example of the use case

- What is the longest duration you might need from a timer?
  - Give an example of the use case

- Concern: high granularity for long durations require MANY bits
  - We often optimize for one of the other

Timer peripheral on nRF52833



Input and Prescaler

$$f_{\text{TIMER}} = \frac{16\text{ MHz}}{2^{PRESCALER}}$$

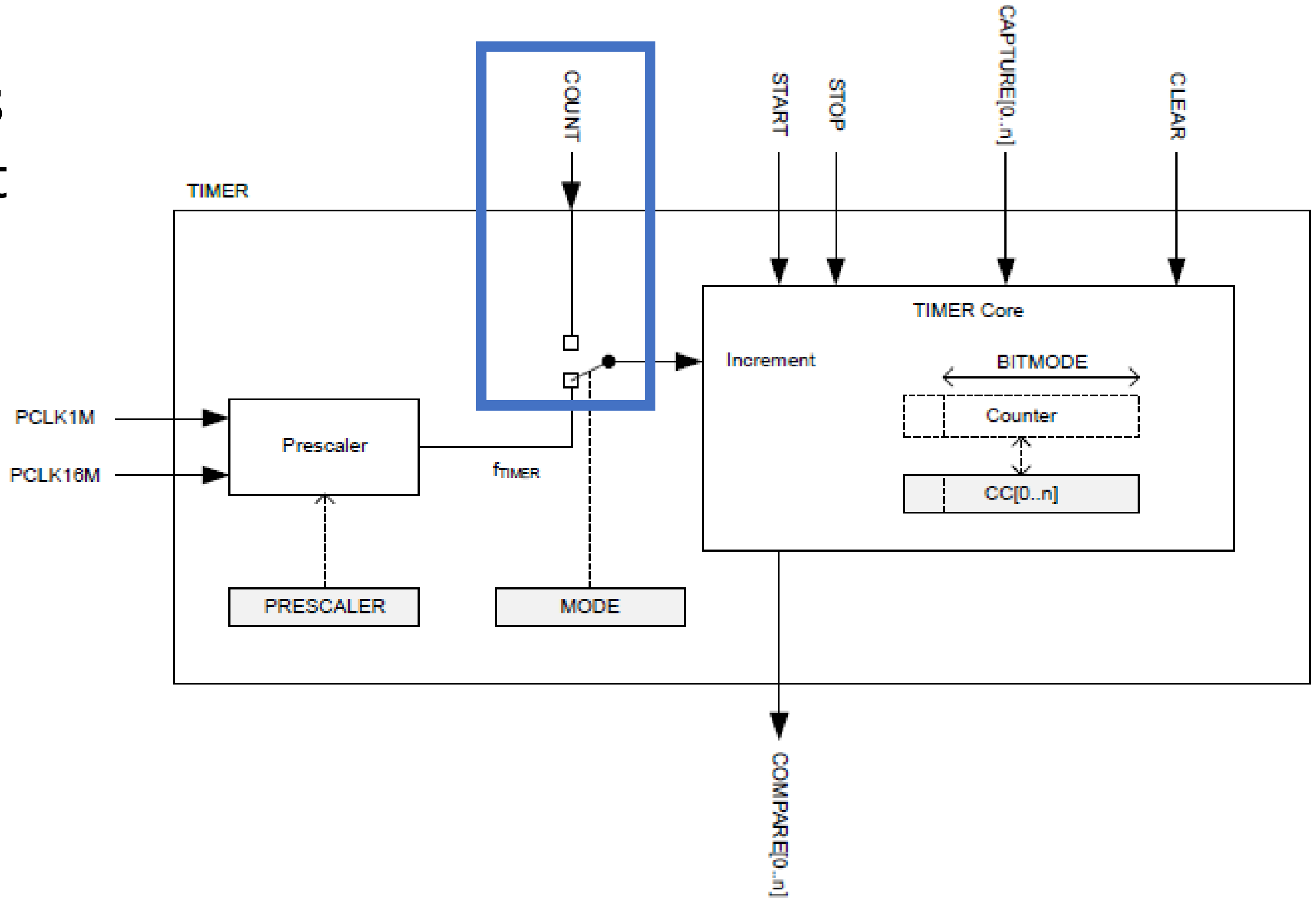
- Prescaler is a 4-bit number
  - Possible timer input clocks: 16 MHz – 488 Hz

- Ticks counted with (up to) 32-bit internal Counter:
  - Minimum
    - **268 seconds** until overflow
    - **62.5 ns** per tick
  - Maximum
    - **101 days** until overflow
    - **2.04 ms** per tick

Alternate input source for counter mode

- Counter mode works with non-timer inputs
  - E.g. GPIO input event

- Count anything!



Capture/Compare registers (CC)

- 32-bit storage registers (each timer has multiple)
  - Uses: capturing or comparing
- On Capture[n] event
  - Internal Counter value copied to CC[n]
  - Then you can read the former Counter value from CC[n]
- Capture used to measure durations of events
  - Capture can be triggered by software or by Events from other peripherals
  - Multiple registers to measure multi-part events

Comparing with CC registers

- When internal Counter value equals a CC register
  - Corresponding Compare[n] event is triggered
  - Can trigger interrupts

- Usually written to in advance to start/stop behavior
  - Toggle LED every second
  - Sample sensor every five minutes
  - Refresh LED matrix every 1/60 seconds

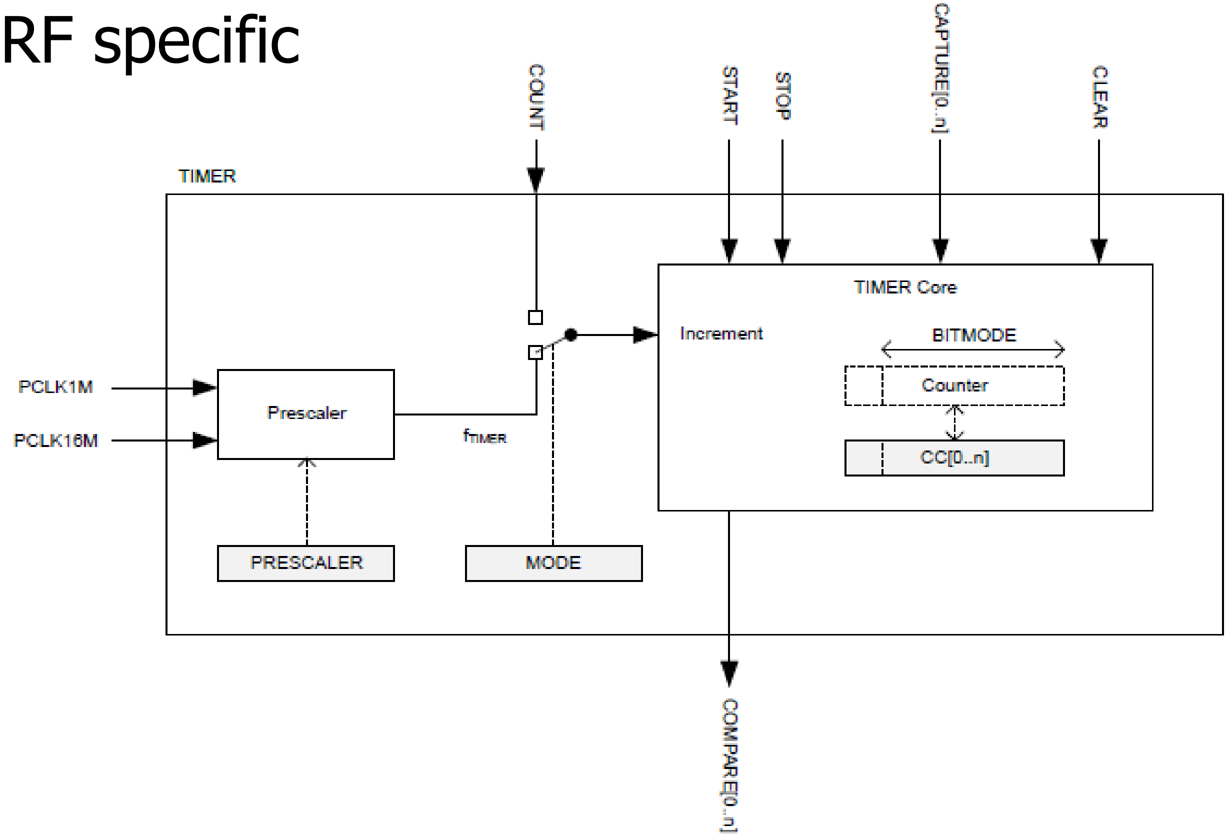
The nRF52833 has multiple Timer instances

6.28.5 Registers

Base address	Peripheral	Instance	Description	Configuration
0x40008000	TIMER	TIMER0	Timer 0	This timer instance has 4 CC registers (CC[0..3])
0x40009000	TIMER	TIMER1	Timer 1	This timer instance has 4 CC registers (CC[0..3])
0x4000A000	TIMER	TIMER2	Timer 2	This timer instance has 4 CC registers (CC[0..3])
0x4001A000	TIMER	TIMER3	Timer 3	This timer instance has 6 CC registers (CC[0..5])
0x4001B000	TIMER	TIMER4	Timer 4	This timer instance has 6 CC registers (CC[0..5])

Bonus concept: shorts

- In a peripheral: **Tasks** are inputs and **Events** are outputs
- Shorts connect an Event to a Task within a peripheral
  - Tasks and Events idea is fairly nRF specific



- Timer shorts
  - Connect Compare[n] to Clear
  - Connect Compare[n] to Stop

Usage: how do we set a one second timer?

- Assume timer is already running

1. Get current time from timer
2. Add 1 second worth of ticks to it
    - $\frac{16000000}{2^{PRESCALER}}$ is the number of ticks per second
3. Set an unused Compare register to value
4. Enable interrupts for that Compare event

Warning: what if you're setting a 1 us timer instead? Or a 100 ns timer?

Timer could expire *before* software writes it to the peripheral.

Break + Check your understanding

- Prescaler value is 4
- Current internal Counter value is 0x1000
- Want a 0.5 second timer

$$f_{\text{TIMER}} = \frac{16\text{ MHz}}{2^{PRESCALER}}$$

- **What do you set the CC[0] register to?** (32-bits)

Break + Check your understanding

- Prescaler value is 4
- Current internal Counter value is 0x1000
- Want a 0.5 second timer

$$f_{\text{TIMER}} = \frac{16\text{ MHz}}{2^{PRESCALER}}$$

- **What do you set the CC[0] register to?** (32-bits)
  - 1 MHz Timer frequency -> 500,000 ticks in 0.5 seconds
  - 500000 -> 0x7A120
  - Plus initial value of counter = **0x7B120**

Outline

- Clocks
- Timers
- **Virtualizing Resources**
- Real-Time Counter
- Watchdog

Choosing resource amounts is a problem

- Problem: applications may require any number of resources
  - Particularly in this case: peripherals
  - For example, how many timers should there be?

- But hardware has to pick some number to provide
  - More is wasted cost
  - Too few and applications cannot succeed

- Solution: virtualize the resource

Virtualization pattern

- Create a queue of requests and a pool of resources
  - N requests to M resources

- Application requests are queued when they come in
  - Rather than serviced immediately

- When a resource is available
  - Pop request from queue (by some priority)
  - Service with hardware
  - Then wait until another resource is available

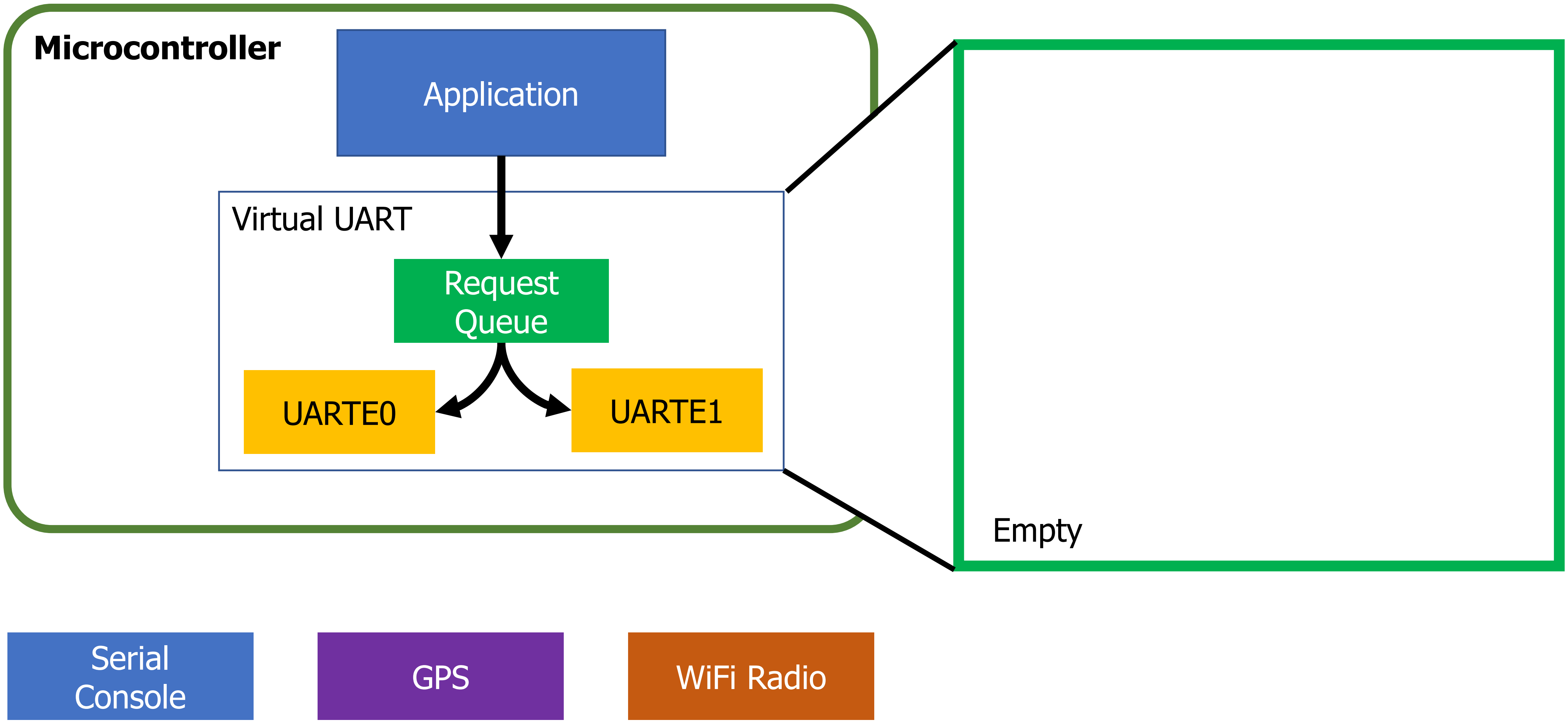
Example: sending serial messages

- Serial messages (such as printf() strings) are sent via UART
  - UARTE peripheral (we'll talk about this later)

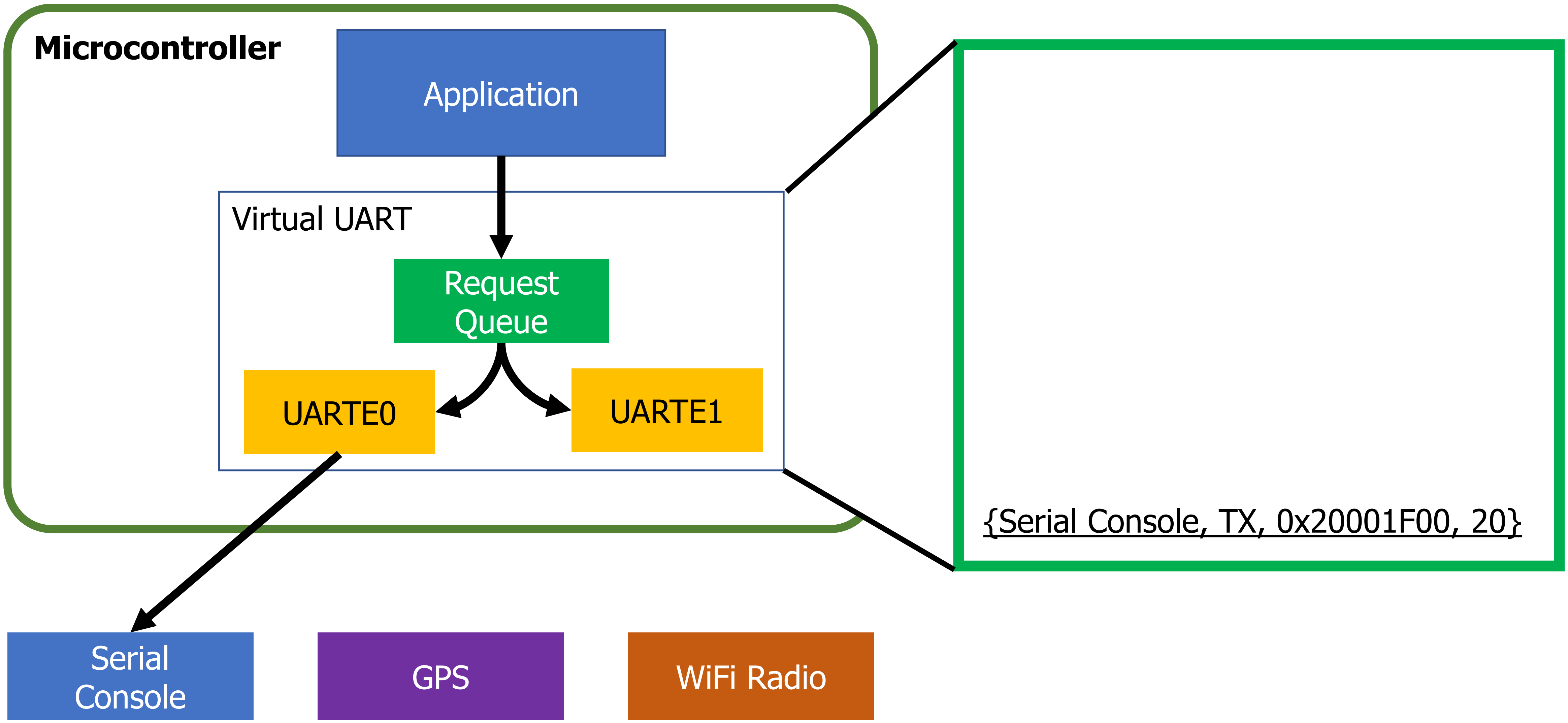
- nRF52 has two UARTE peripherals
  - Can be attached to any output pins
  - Changing pins is a quick operation

- What if we want to talk to three serial devices?
  - Console (printf output)
  - GPS (NMEA)
  - WiFi radio (AT commands)

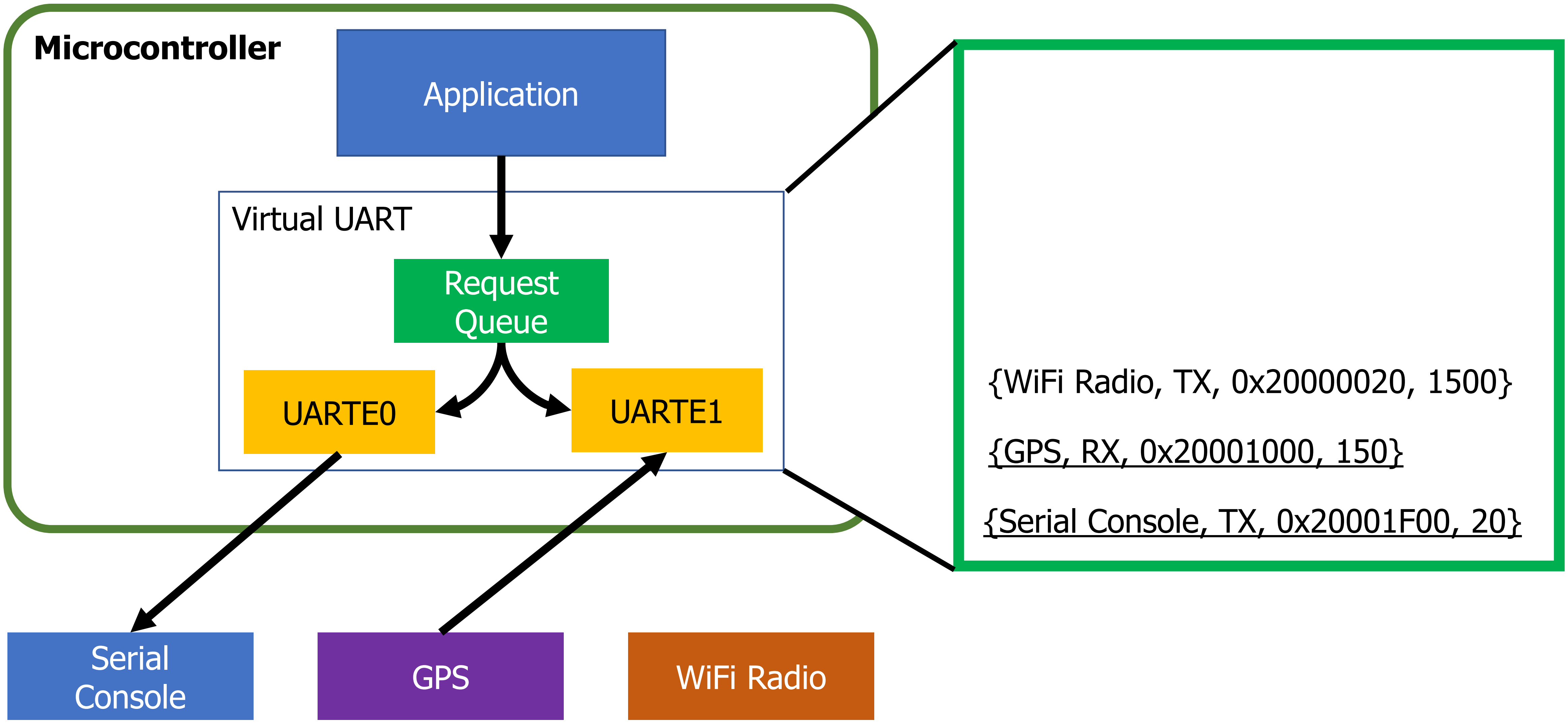
Virtualized UART



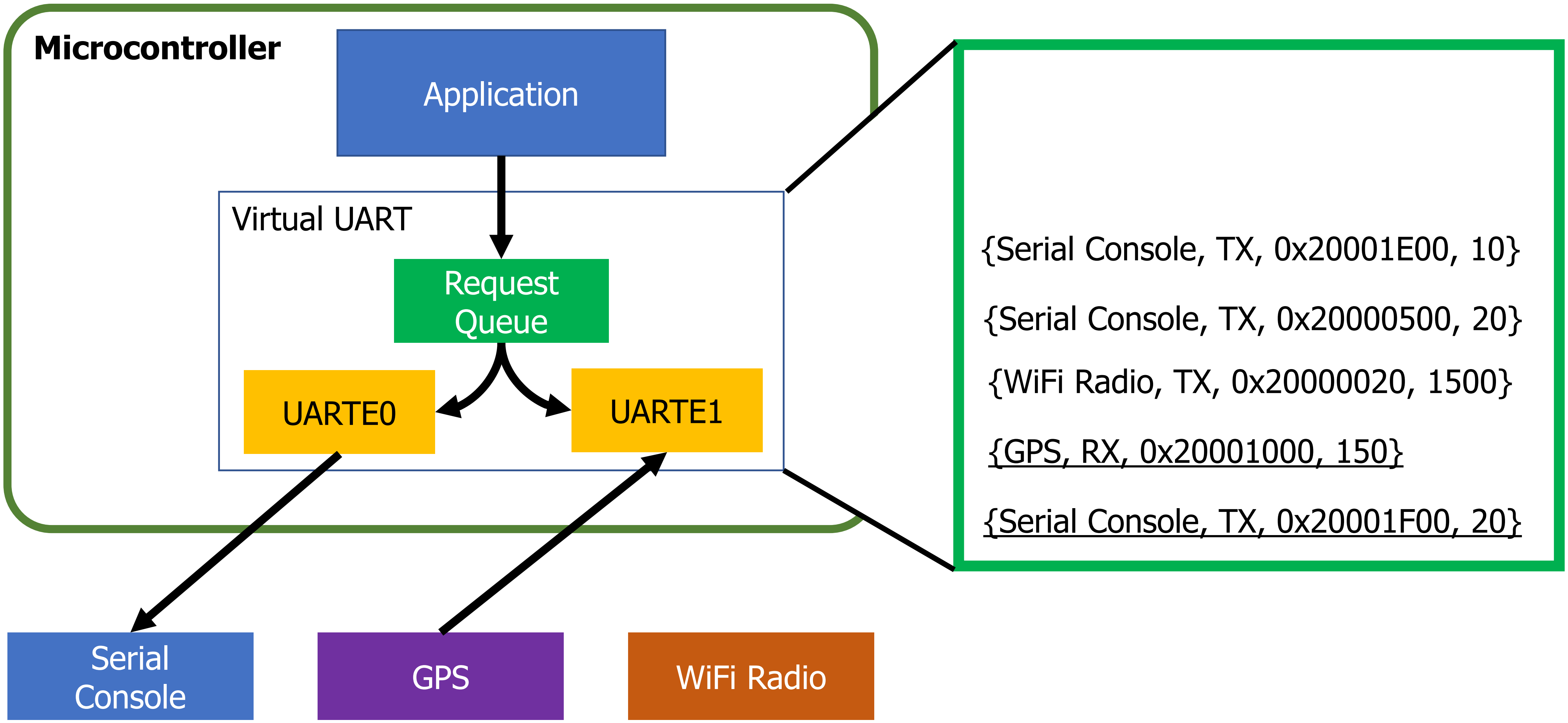
Virtualized UART: serves request with hardware



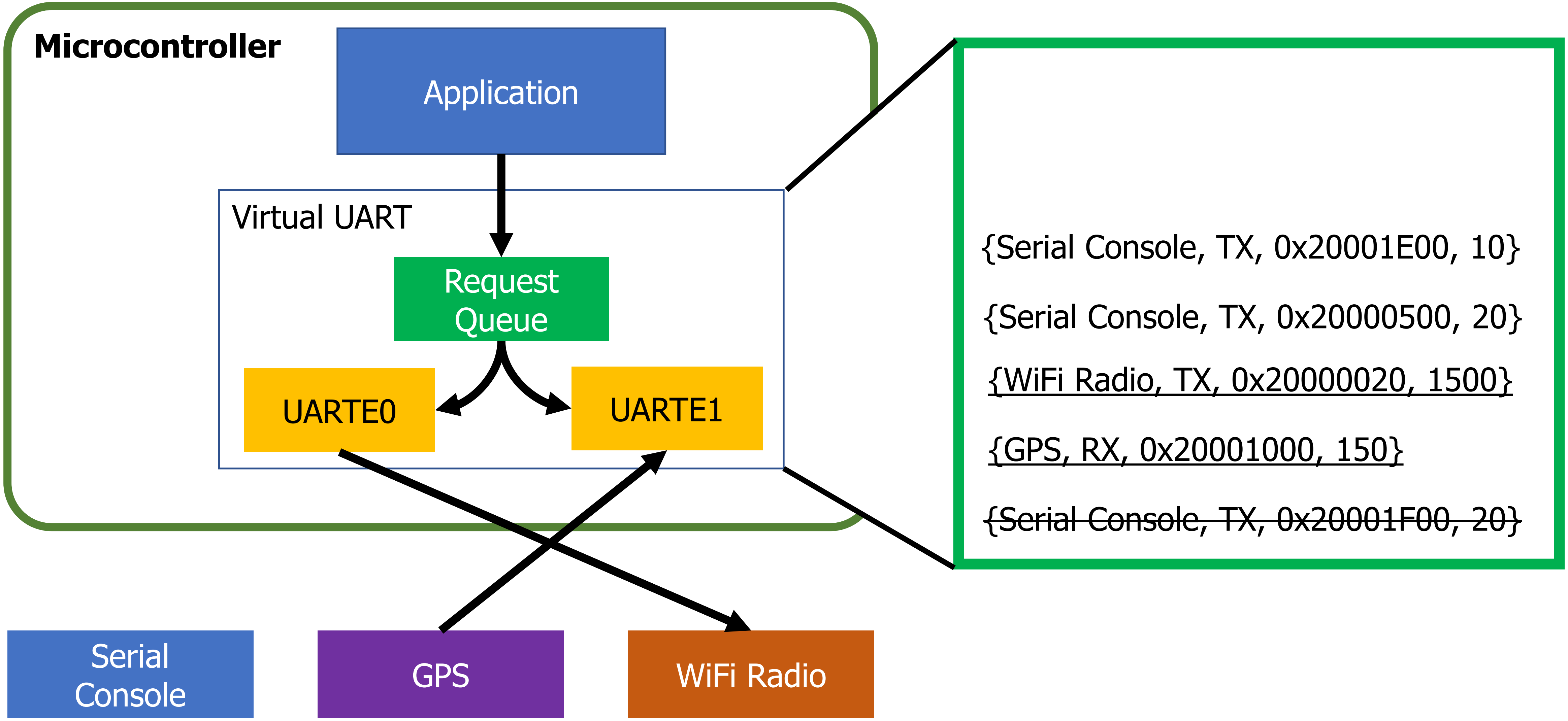
Virtualized UART: serves until resources are full



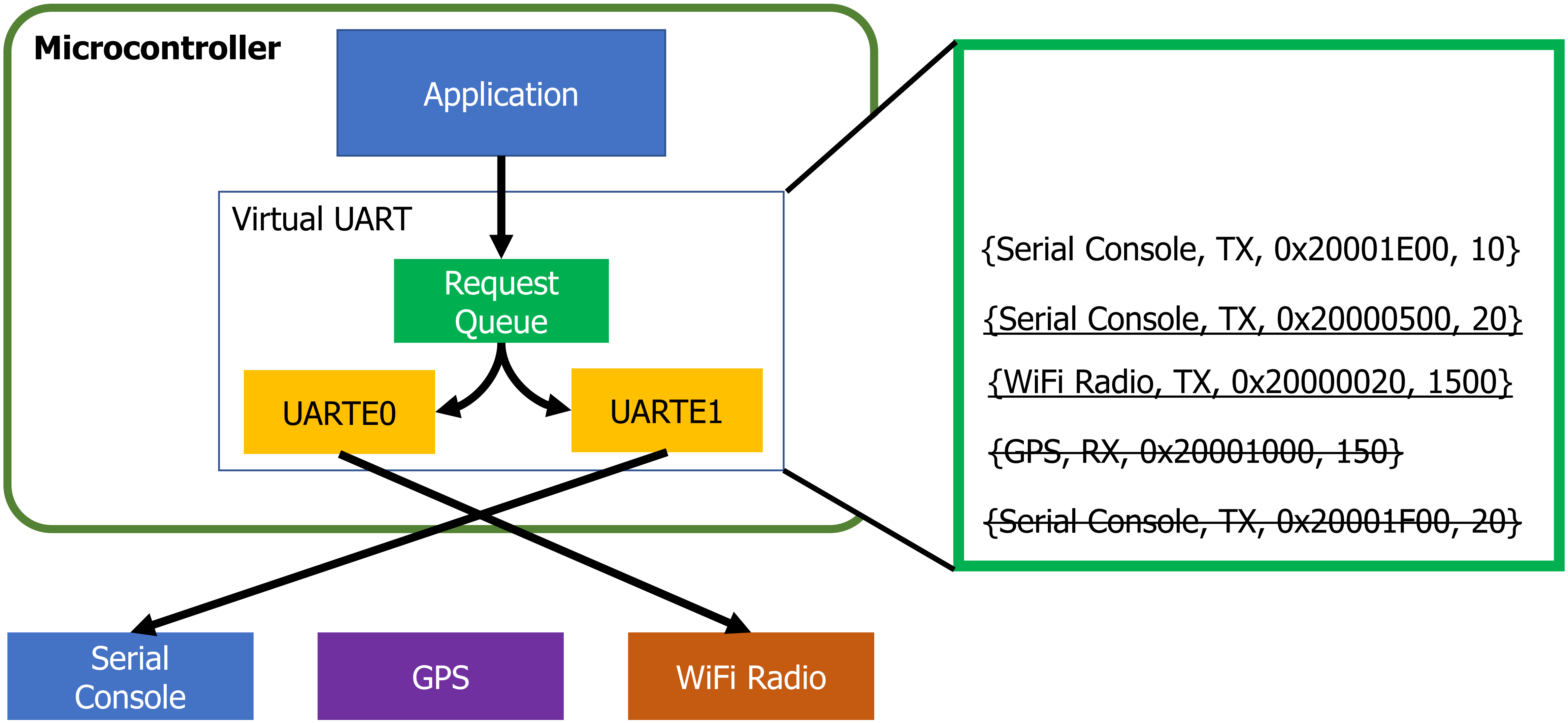
Virtualized UART: additional requests are queued



Virtualized UART: moves to next item when complete



Virtualized UART: moves to next item when complete



Challenges to making virtualization work

- How fast are requests coming in?
    - Requests more quickly than service are an unsatisfiable system
- How long does it take to reconfigure the resource?
    - Long delays could mean high latency
    - Might want to optimize for requests with same configuration first
- Need to ensure all of the configuration changes
    - Common bug: forget to modify part of one register and system works most of the time, but not in all cases
- Need ability to queue requests
    - Usually stored in a linked list structure
    - Dynamically… But we generally want to avoid dynamic memory

Dynamic resource allocation options

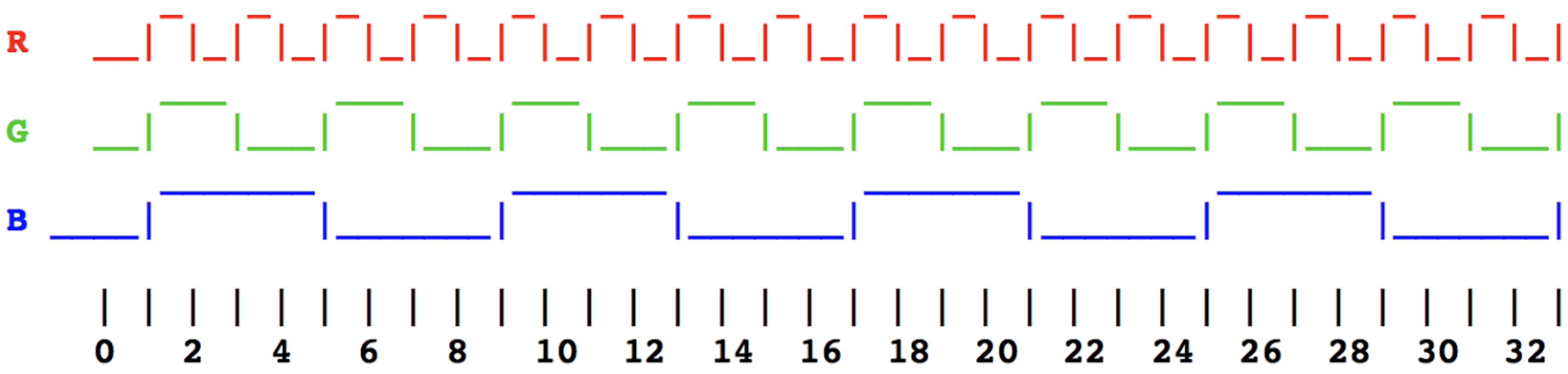
1. Create a queue with a maximum size in Virtual Driver
    - Some number larger than the hardware picked, based on app knowledge
    - Still either runs out or wastes memory

2. Just use malloc()
    - Is actually possible on the nRF52 with newlib (libc implementation)
    - Might run out, but then just wait for requests to complete

3. Create list nodes individually as global variables
    - Application decides how many it needs at compile time
    - Passes them into the Virtual Driver at first use
        - "Here's my request and a linked list node to store it in"

Another example: managing multiple timers

- You often have tasks that look like this:



- Most easily thought about as three separate timers
    - But maybe the system doesn't have that many timers to spare!
    - Virtualization can help

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests
1. 10010, A
2. 10050, B
3. 10110, C
4. 20000, D

CC Register: 10010

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests
1. 10010, A
2. 10050, B
3. 10110, C
4. 20000, D

CC Register: 10010

Call timer handler A!
Update CC register and list

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests

CC Register: 10050

1. 10050, B
2. 10110, C
3. 20000, D

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests

1. 10050, B
2. 10110, C
3. 20000, D

CC Register: 10050

Call timer handler B!
Update CC register and list

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests **CC Register:** 10110

1. 10110, C
2. 20000, D

Virtual timers

- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests

1. 10100, E
2. 10110, C
3. 20000, D

CC Register: 10100

New request arrives for 10100
Enqueue and sort queue
Update CC if first request has changed

Enqueuing timer requests

- Timer requests come in the form: {N seconds from now}
  - timer_request(duration, handler);
- Requests are always relative to the current time
- Need to enqueue by expiration time
  - Duration + Current Time
  - Allows for a globally sortable list
    - Need to decide how to handle overflow logic in real world

Make sure not to miss timers

- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event
  - Check for this, and call handler manually if necessary

Timer Requests

1. 10100, E
2. 10110, C
3. 20000, D

CC Register: 10100

Handle 10100 event, Call E

Make sure not to miss timers

- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event

Timer Requests

1. 10110, C
2. 20000, D

CC Register: 10110

Update list
Update CC register
Oh no! That’s in the past!!

Break + Question

- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event

- What do we do about the missed timer?
  - There are multiple “correct” answers here

Break + Question

- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event

- What do we do about the missed timer?
  - There are multiple “correct” answers here

- **Some options:**
  - Just call handle that timer event as soon as possible
    - Possibly telling it about the delay

  - Crash the system! (Deadlines cannot be missed in some systems)
    - Or at least enter some fault recovery handler

Make sure not to miss timers

- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event
  - Check for this, and call handler manually if necessary

Timer Requests

1. 20000, D

CC Register: 20000

Call C manually
Update list and CC register again

Some timers are periodic

- Repeating timers are easy to add to this system
  - Include a Boolean for “repeating” and the duration in the request

- When timer expires
  - If not repeating, just call handler and then drop it
  - If repeating,
    - First reinsert based on duration and new current time
    - Then call the handler
      - Don’t want the latency of the handler to slow us down

Concurrency safety

- Modifying the request structure in an interrupt context is dangerous
  - New request might be in the middle of getting added
  - Interrupt would run right in the middle of that
  - Literally an OS data race example

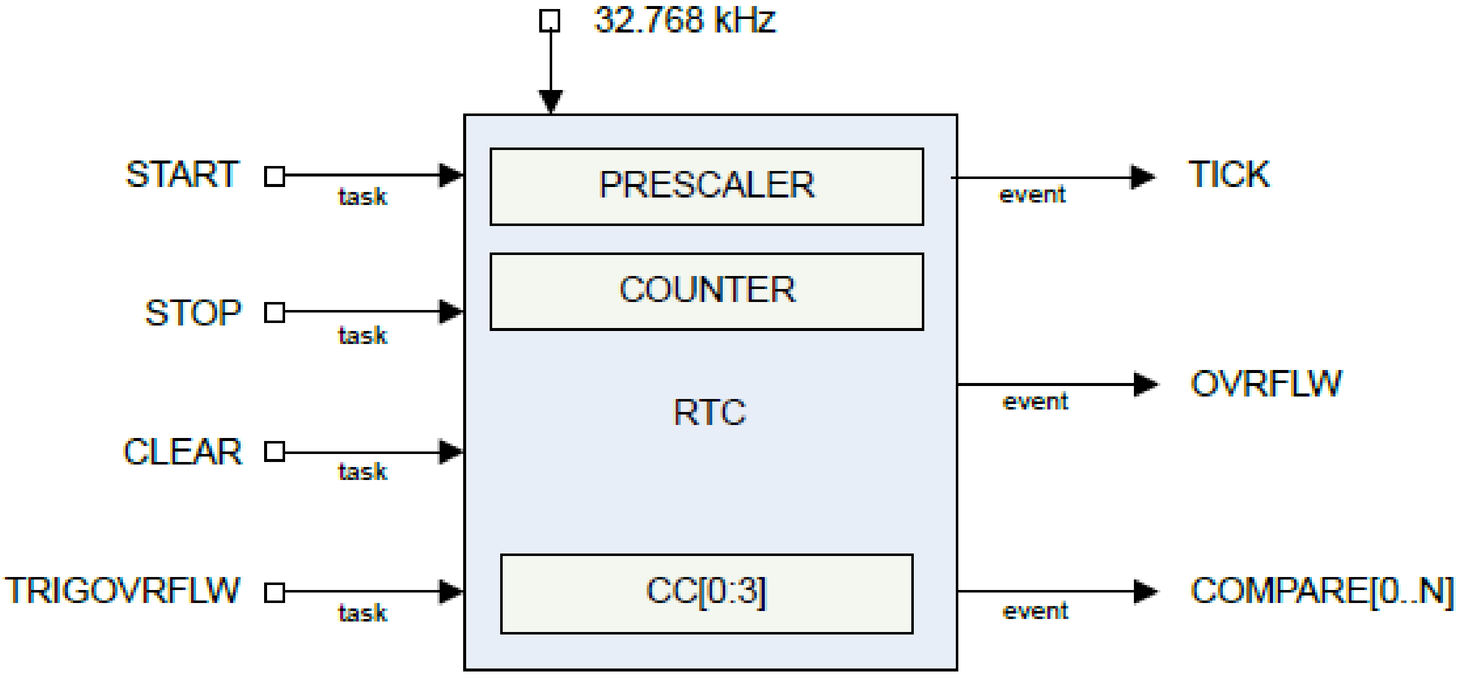
- Solution: disable interrupts during critical section
  - Whenever editing request structure
  - Enable interrupts after, which may result in an event
    - Note: Interrupt handler might now fire but have no work to do. Should always check if something should actually be handled first

Outline

- Clocks
- Timers
- Virtualizing Resources
- **Real-Time Counter**
- Watchdog

Real-time Counter

- Low-power (32 kHz) version of Timer
  - Only a 24-bit internal Counter



- Note: abbreviated RTC, but that already means something else (Real-Time Clock)

Differences between Real-Time Counter and Timer

- Runs off of LFCLK instead of HFCLK
  - With smaller prescaler value (4096 vs 32768)

- 24-bit counter vs 32-bit counter for Timer

- Can read the Counter value directly
  - No need for Capture task

- Otherwise extremely similar. Just a low-power version of Timer

Time resolution for Real-Time Counter

$$f_{\text{TIMER}} = \frac{32\ \text{KHz}}{Prescaler+1}$$

- Resolution
  - Minimum: 30.517 μs ticks, overflows in 512 seconds (24-bit Counter)
  - Maximum: 125 ms ticks, overflows in 582 hours
- Not as precise as the Timer (which has 62.5 ns best precision)
  - Possible design: use both
    - Real-Time Counter for most of the waiting
    - Chained into Timer for precise remaining amount of time

Comparing timer types

- Real-Time Counter
  - Low precision and duration
  - Low energy

- Timer
  - High precision or duration
  - High energy

nRF SDK Virtualized Timers: APP_TIMER

- Runs off the RTC
- `APP_TIMER_DEF` creates a node for the timer and initializes it
- `app_timer_create` inserts the node in an internal linked list
- `app_timer_start` actually starts running the timer
- [SDK documentation]()

```
// Create a new timer instance
APP_TIMER_DEF(my_timer);

int main(void) {
  // Initialize the timer library
  app_timer_init();

  // Initialize a timer instance
  // Mode: single or repeated
  // Callback function: called on expiration
  app_timer_create(&my_timer,
                   APP_TIMER_MODE_REPEATED,
                   callback_function);


  // Start a timer
  // Duration: 32768 ticks per second
  //
  app_timer_start(my_timer, 32768, NULL);
}
```

For example code, in nu-microbit-base
see: apps/app_timer_example/main.c

Outline

- Clocks
- Timers
- Virtualizing Resources
- Real-Time Counter
- **Watchdog**

Reliable systems

- What’s the most common way to solve computer problems?
  - Turn it off and turn it on again.

- **Why?**

Reliable systems

- What’s the most common way to solve computer problems?
  - Turn it off and turn it on again.

- **Why?**

- Resets “state” to original values, which are likely good
  - Startup is often well-tested
  - It’s long-running code interacting in unexpected ways that leaves systems in a broken state

Watchdog timer (WDT)

- Focused on failures where the system “hangs” forever
  - Maybe software, maybe hardware!

- Can’t know for certain the system is hung, but can know practically
  - Select a timeout that is the maximum amount of time you expect the system to ever go without looping in main()
  - Multiply it by 2-10
  - Set a watchdog timer to that value

- If watchdog timer ever expires, it resets the system (in hardware)

Watchdog configuration

$$\text{timeout (seconds)} = \frac{Counter\ Reload\ Value + 1}{32768}$$

- Configure watchdog
  - Can choose whether to count down during Sleep mode or Debug mode

- Set a Counter Reload Value (CRV, 32-bits)

- Start the watchdog timer
  - Loads internal Counter to CRV value
  - Starts counting down at 32 kHz

Running applications with a watchdog timer

- Need to periodically reset the watchdog to keep it from expiring
  - Known as “feeding” the watchdog or “kicking” the watchdog
- Reload Request register
  - Must write sequence 0x6E524635 to reload watchdog (“nRF5”)
  - Incredibly unlikely to happen by accident
- While running, watchdog is protected from modification
  - Configure once, run forever (at least until a reboot)
  - Only option is to make periodic Reload Requests
- Default off on the nRF52833 (default on for the MSP430!)

Outline

- Clocks
- Timers
- Virtualizing Resources
- Real-Time Counter
- Watchdog