Lecture 07
Driver Design

CE346 – Microprocessor System Design

Branden Ghena – Spring 2021

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

Today’s Goals

- Explore two aspects of device driver design
  - Virtualization
  - Non-blocking vs Blocking interfaces

- Discuss how interrupts interact with these
  - Event-loop as a partial alternative

Outline

- **Virtualized Drivers**
- Driver Interfaces (Blocking and Non-Blocking)
- Event Loop

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
- While a resource is available
  - Pop request from queue (by some priority)
  - Service with hardware
  - Wait until another resource is available

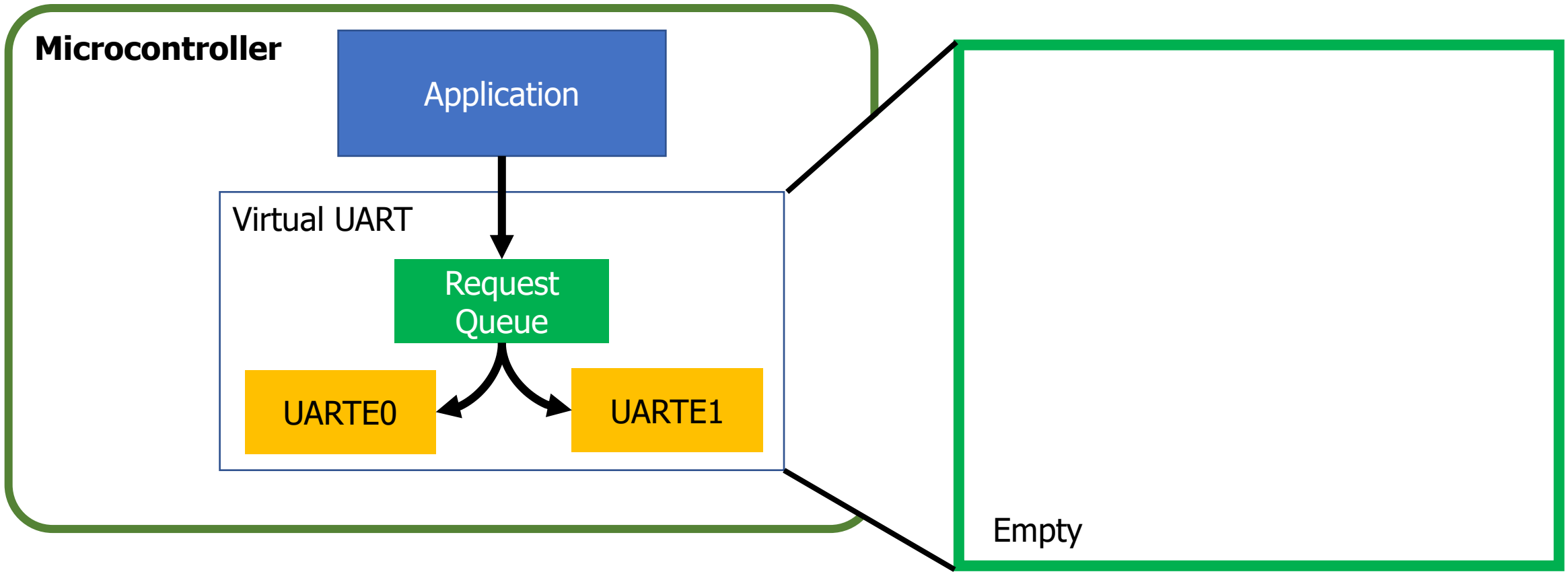
Example: sending serial messages

- 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

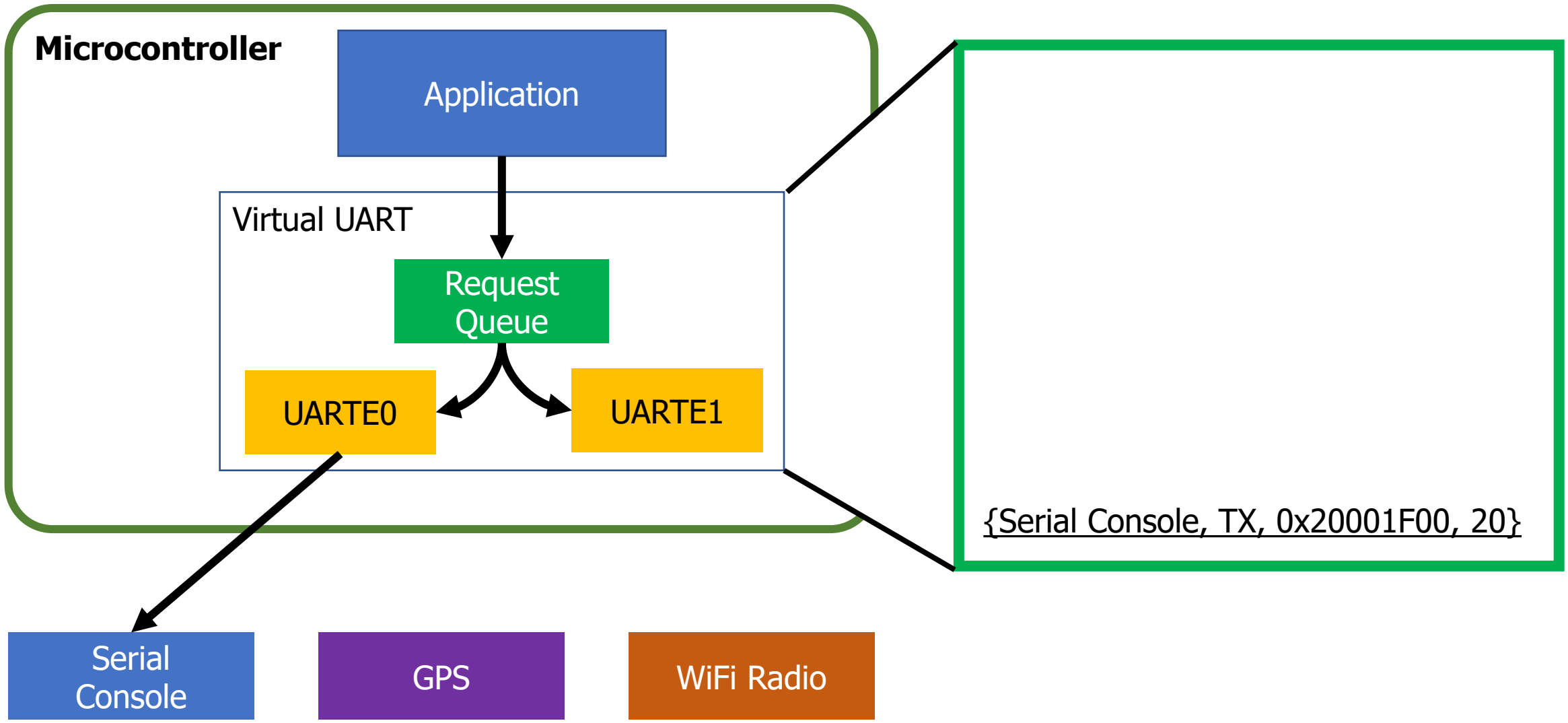


Serial Console

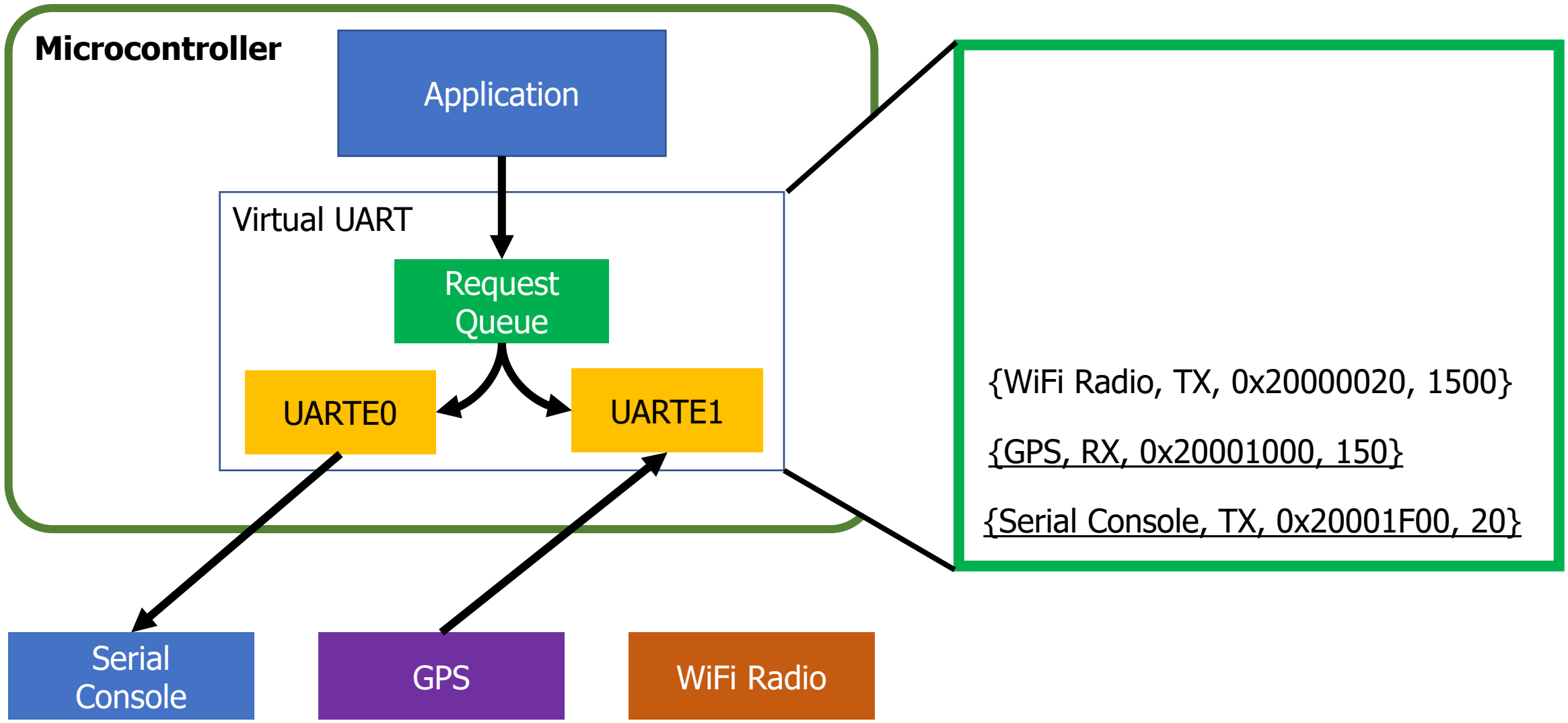
GPS

WiFi Radio

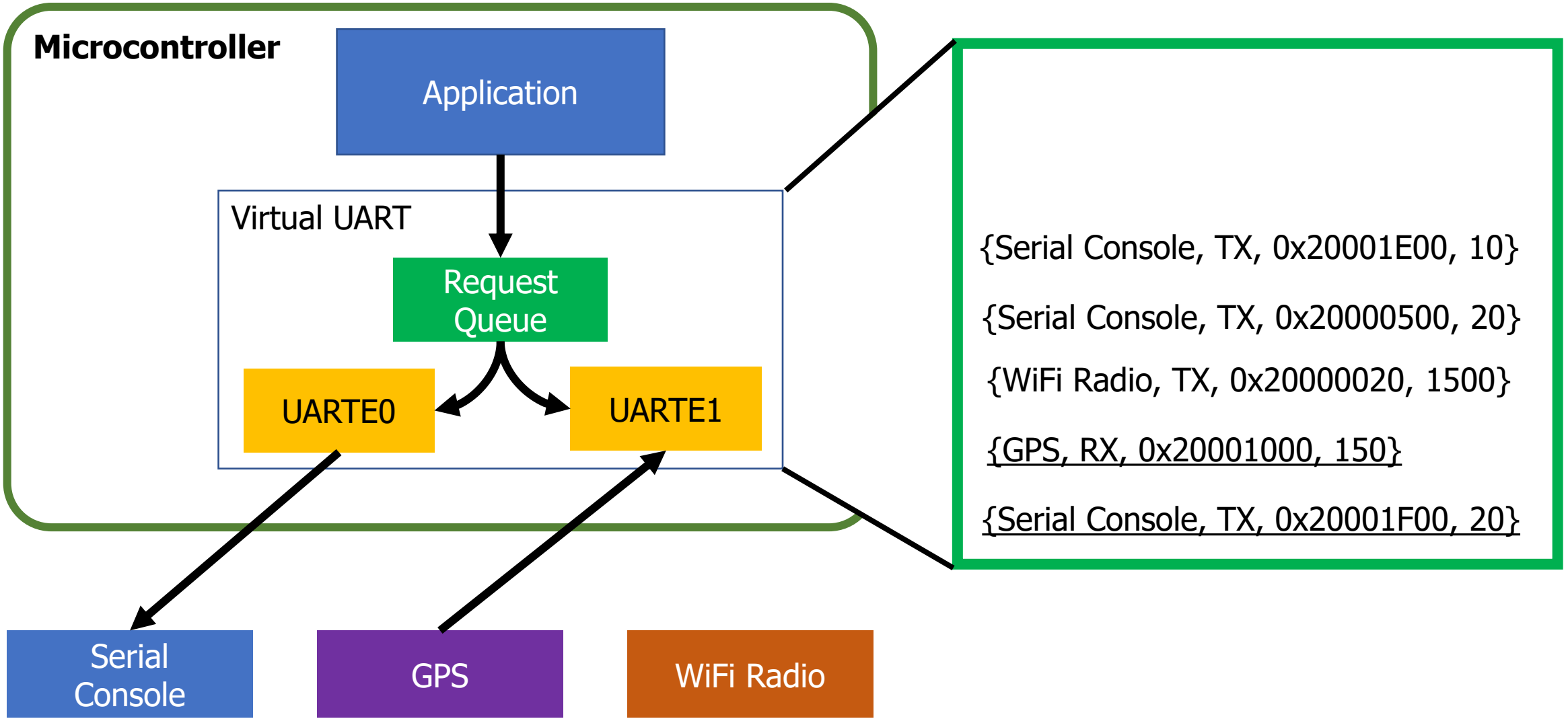
Virtualized UART: serves request with hardware



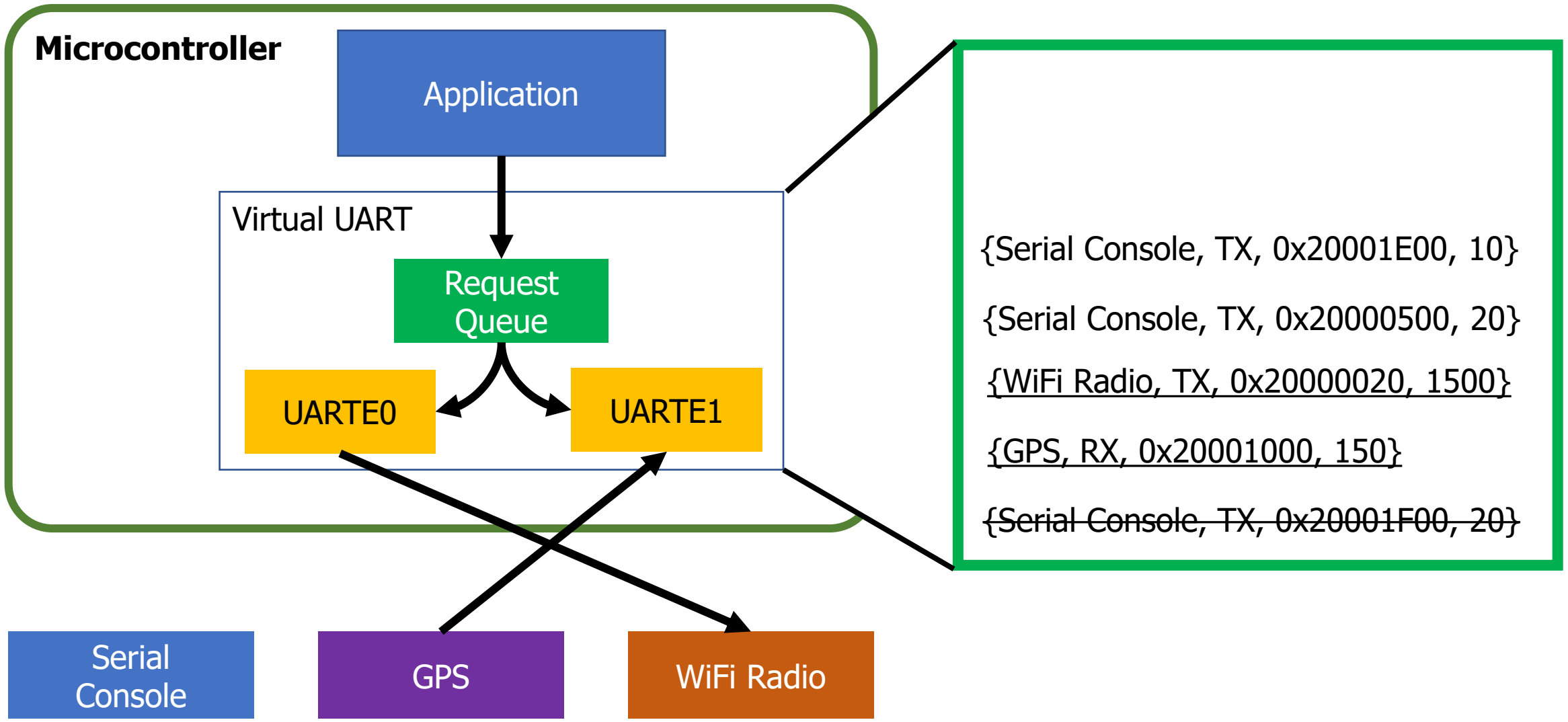
Virtualized UART: serves until resources are full



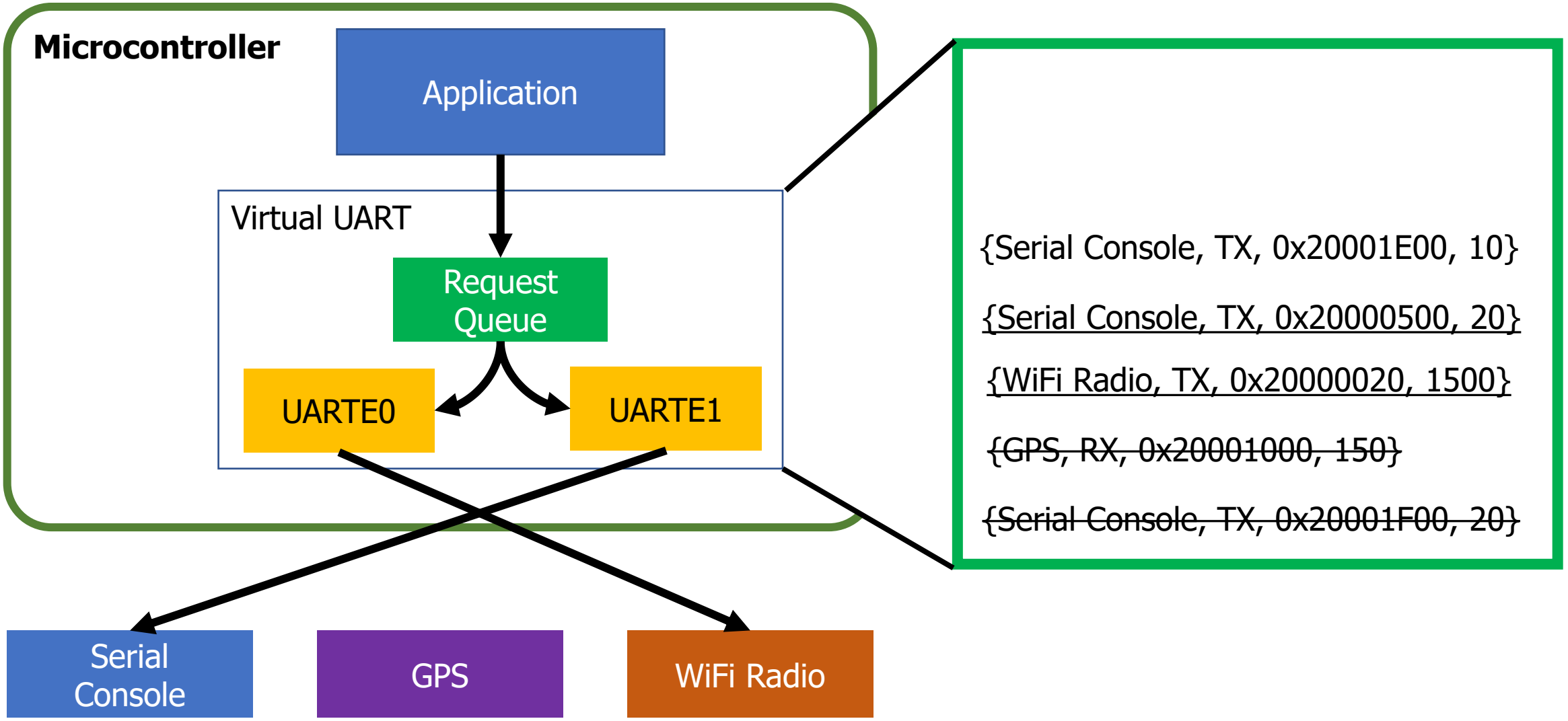
Virtualized UART: additional requests are queued



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

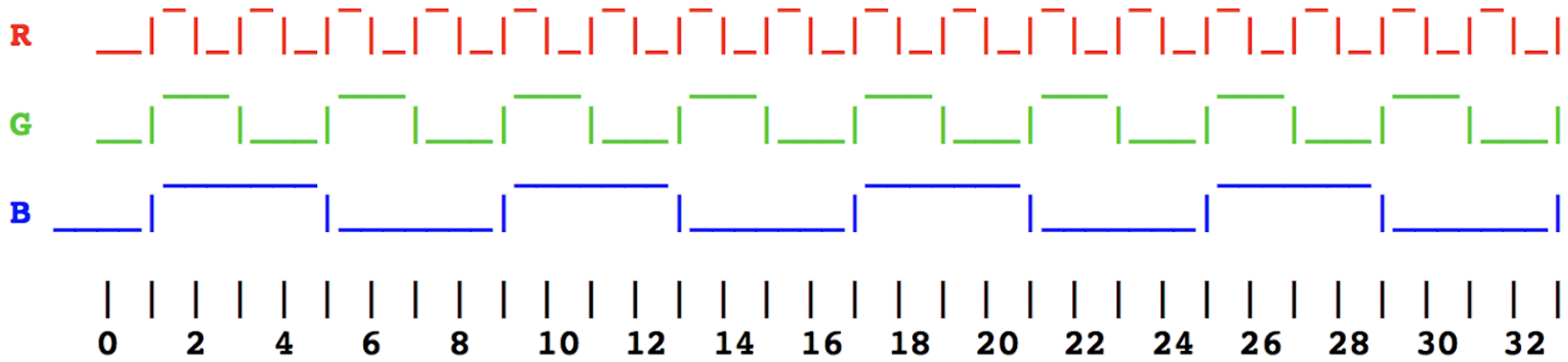
1. Create a queue with a maximum size in Virtual Driver
   - Some number larger than the hardware picked, based on app knowledge
   - Still either runs out or wastes memory

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

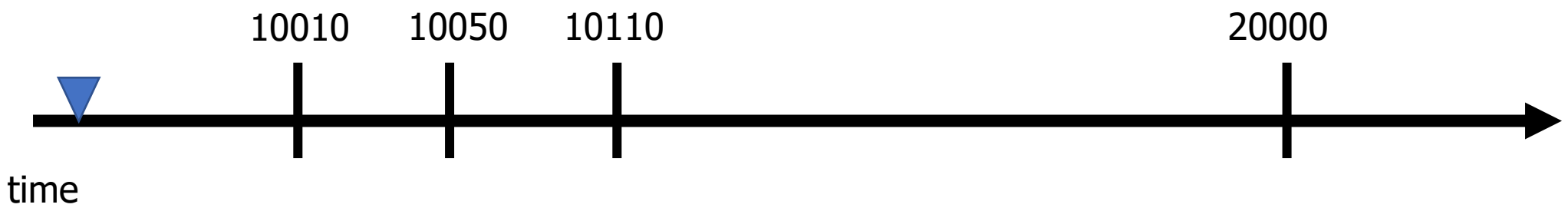
Virtual timers

- 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
2. 10050
3. 10110
4. 20000

CC Register: 10010



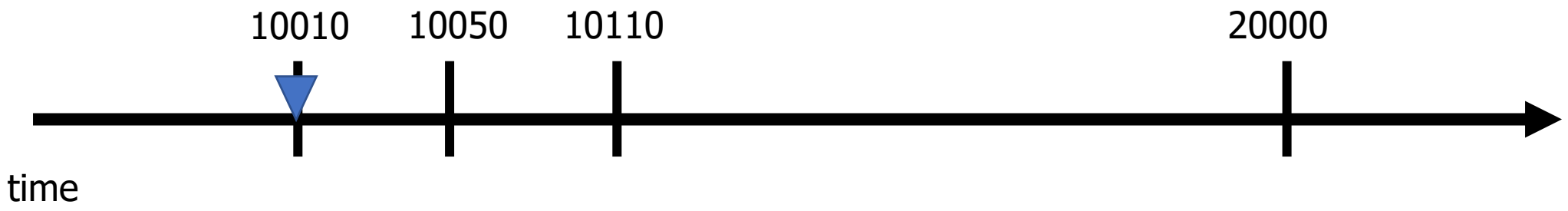
Virtual timers

- 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



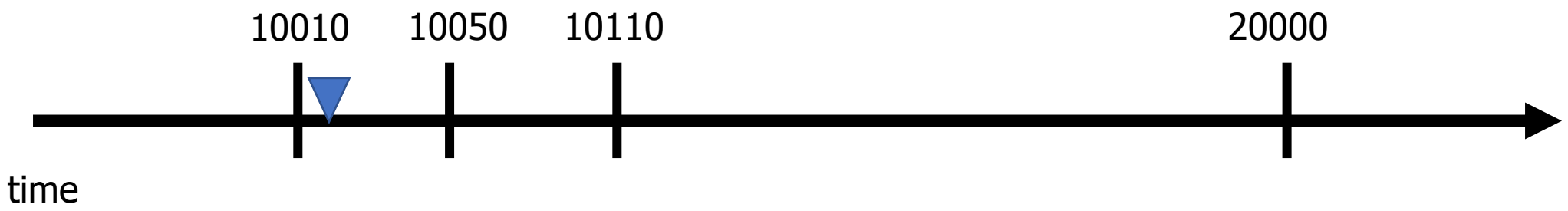
Virtual timers

- 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

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

CC Register: 10050



Virtual timers

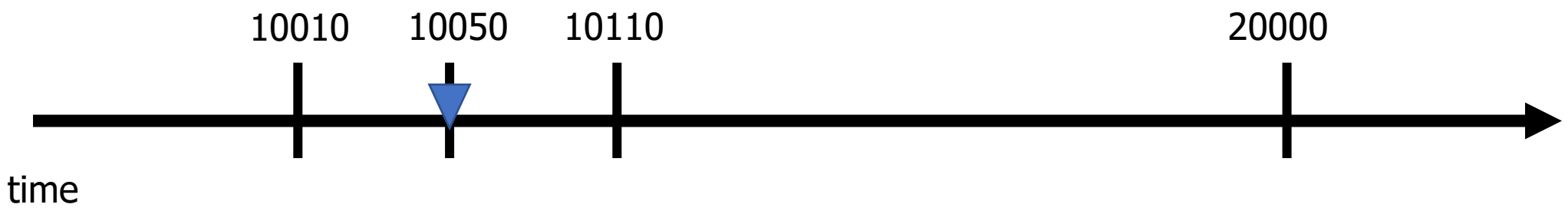
- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

Timer Requests

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

CC Register: 10050

Call timer handler B!
Update CC register and list

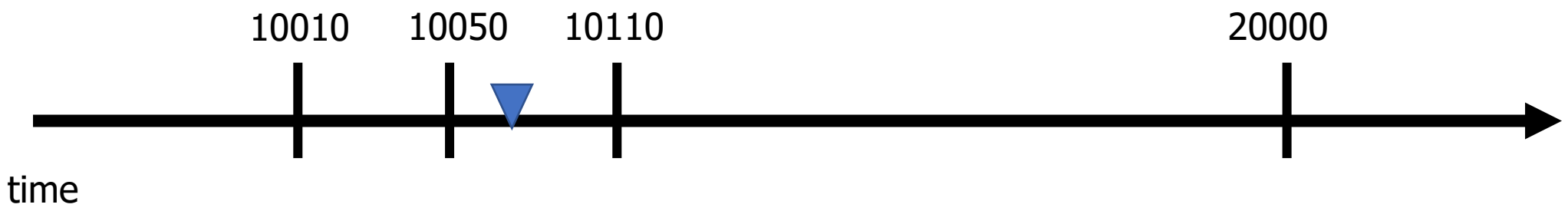


Virtual timers

- 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

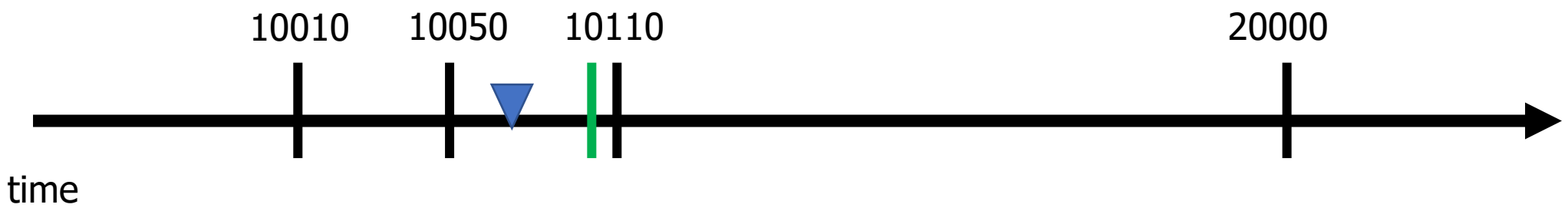
- Solution: keep a list of timer expiration times
  - Soonest expiration goes in the Capture/Compare register
  - Others stay in linked list, sorted by expiration

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

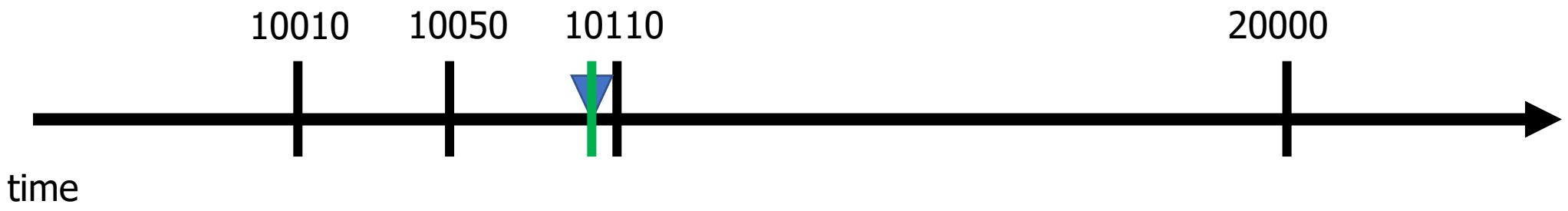
- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event
  - Check for this, and call handler manually if necessary

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
  - Check for this, and call handler manually if necessary

Timer Requests

1. 10110, C
2. 20000, D

CC Register: 10110

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

Make sure not to miss timers

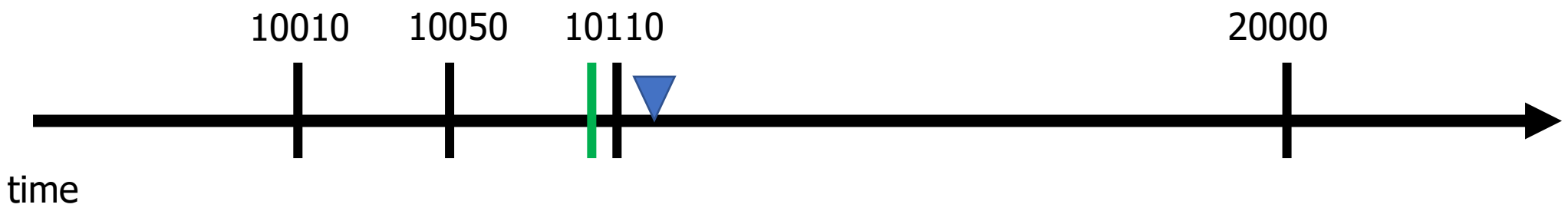
- Sorting list and modifying the CC register takes time
  - Might have skipped right past the soonest event
  - Check for this, and call handler manually if necessary

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

- Virtualized Drivers
- **Driver Interfaces (Blocking and Non-Blocking)**
- Event Loop

Callback functions

- `timer_start(duration, my_timer_handler, context);`

- Driver interfaces often provide a callback mechanism
  - Caller provides a function which should be executed when complete

- "Context" is often provided as well (void*)
  - Ability for caller to pass an argument for the callback function
  - Often a pointer to a position in a structure or a shared variable to modify

Function pointers in C

- Harder than in Javascript or C++. Can't define anonymous function inline
  - Instead create a pointer to an existing function in your code

```
void myfun (int a) {
    // do something here
}

void main() {
    void (*fun_ptr)(int) = &myfun;
    fun_ptr(10); // dereference happens automatically
}
```

Callbacks usually run in an interrupt context

- If the interrupt handler calls the callback, the callback will be within that same interrupt context

- Be careful which variables you modify!!
  - Same concurrency problems mentioned before

- Starts to get pretty annoying
  - Embedded systems deal with concurrency issues just like OS

Blocking function calls

- Alternative option: blocking calls
  - Do not return until request is complete

```
void myfun (void* context) {
    *(boolean*)context = true; // context is the flag pointer
}

void timer_start_blocking(duration) {
    boolean flag = false;
    //          duration, pointer, context
    timer_start(duration, &myfun, &flag);
    while (!flag) { }
}
```

Temp driver example

nu-microbit-base/software/apps/temp_driver/

Outline

- Virtualized Drivers
- Driver Interfaces (Blocking and Non-Blocking)
- **Event Loop**

Interrupts are frustrating

- We do not want to block on every call
- We also do not want to deal with concurrency issues
- Alternative: one main event loop
  - Polls necessary sensors
  - Iterates through state machine and determine actions
  - Runs at a certain frequency

Event loop

- Rather than polling a single driver, poll all of them
  - Each time through the loop check all relevant inputs
  - Respond to events that are necessary
  - Sleep until ready to start again

```
while (1) {
    time start = get_time();
    boolean result = check_timer();
    if (result) { check_gps(); }
    adjust_throttle();
    sleep(1ms - (get_time() - start));
}
```

Top-half / Bottom-half handler design

- Top half
  - Implements interface that higher layers require
  - Performs logic to start device requests
  - Wait for I/O to be completed
    - Synchronously (blocking) or asynchronously (return to event loop)
  - Handle responses from the device when complete

- Bottom half
  - Interrupt handler
    - Continues next transaction
    - Or signals for top half to continue (often with shared variable)

Outline

- Virtualized Drivers
- Driver Interfaces (Blocking and Non-Blocking)
- Event Loop