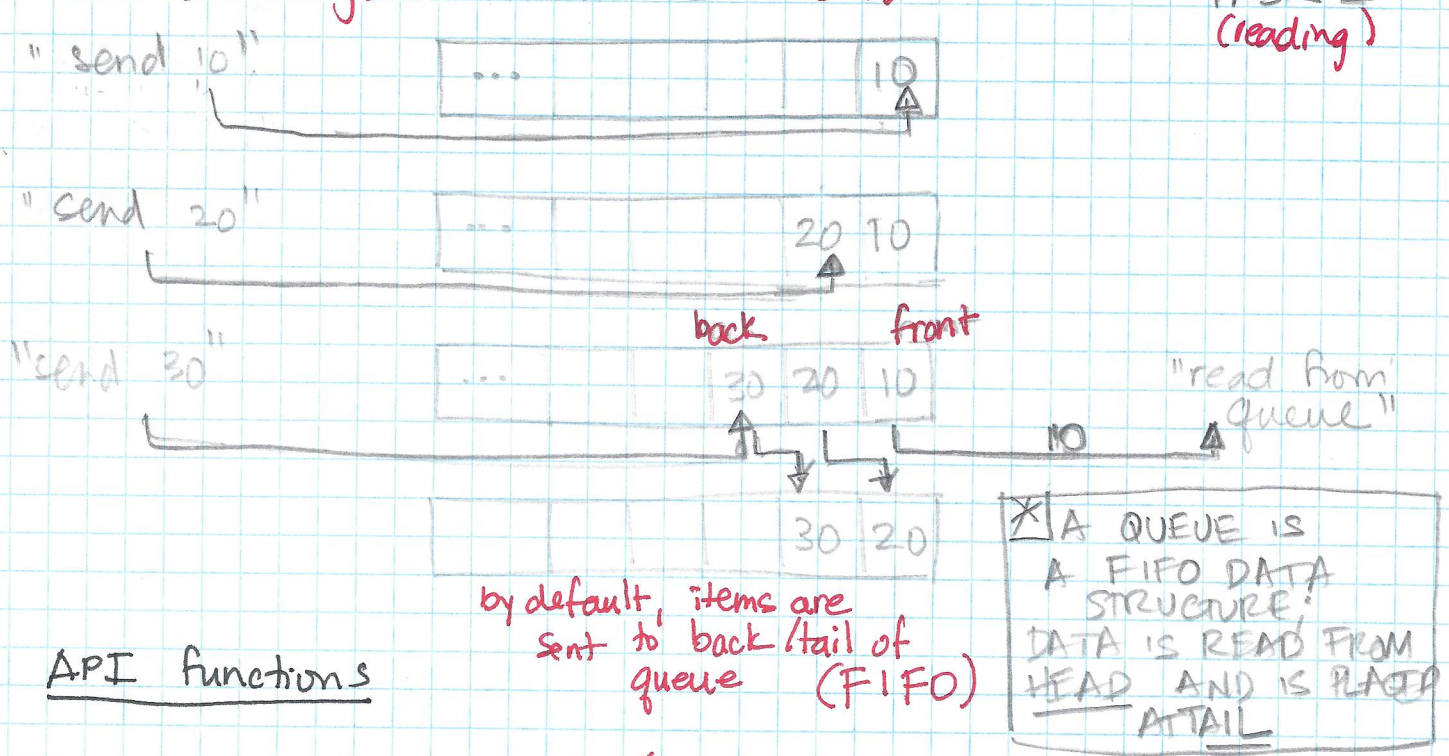
FreeRTOS Queues & Mutexes

- Please have students read corresponding chapter from Richard Barry's book.



A QUEUE IS A FIFO DATA STRUCTURE. DATA IS READ FROM HEAD AND IS PLACED AT TAIL

API functions

- QueueHandle_t xQueueCreate(portBaseType_t uxQueueLength, portBaseType_t uxItemSize);

  type names can change, depending on port

- BaseType_t xQueueSendToFront(QueueHandle_t xQueue, const void* pvItemToQueue, TickType_t xTicksToWait)

- BaseType_t xQueueSendToBack(QueueHandle_t xQueue, const void* pvItemToQueue, TickType_t xTicksToWait)

- BaseType_t xQueueReceive(QueueHandle_t xQueue, const void* pvBuffer, TickType_t xTicksToWait)

- UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);

Mutex (Binary Semaphore)

- used for protecting a resource from access by more than one task at a time
- see: PRIORITY INVERSION
  (when a task of lower priority is executing in spite of the fact that higher priority tasks are in the READY state)

→ EXERCISE:
- have students read Mutex-related material and compile their own lists of relevant API functions.

needed elements to use a mutex:

#include "semphr.h"

type: xSemaphoreHandle

API functions (students should find themselves)
- xSemaphoreTake()
- xSemaphoreGive()
- xSemaphoreCreateMutex()

Real-time Resources

- A number of system resources must be sized and managed in any real-time embedded system, including:

  PROCESSING: the number and type of cores available

  MEMORY: all volatile and non-volatile storage elements

  I/O: buses, digital communications (serial/parallel) used for sensor information decoding and encoding of actuator data
  - includes interconnections between cores.

- traditionally, the focus of real-time resource analysis and theory has centered around processing and executing multiple services on single core.

- resources are allocated to specific threads of execution; the mechanics of preempting a running thread, saving its state, and dispatching a new thread is called a thread context switch.

- Scheduling ⟷ implement a policy } how the RTOS makes a decision

  preemption, dispatch } ⟷ context-switching mechanisms } how the RTOS implements its policy

- Some important factors affecting real-time resource allocation:
  - SPEED or clock rate ([GHz])
  - EFFICIENCY or clocks per instruction (CPI)
    or instructions per clock (IPC) → relevant in pipelined execution

- Algorithm complexity:

  C_i = instruction count on longest execution path for service i, and ideally, deterministic (known a priori). If not known exactly, a WCET should be used (worst-case execution time).

- Frequency of Service Requests, F_i

  $T_i = \frac{1}{F_i}$ = "service release period"

- **Latency Issues**

  - often associated with I/O, interconnect:
    - arbitration (contention) latency for shared I/O devices
    - read latency
    - transit time from device to CPU core
    - registers, tightly coupled memory (TCM), L1 cache for zero wait state
    - single cycle access
    - read/write latencies

  L1 cache is memory on the main CPU chip

- **Bandwidth**
  - average bytes or words transferred per unit time
  - can be "uniform" or "busty"

- **Queue Depth**
  - queuing can keep the system running well, as long as they are not filled → which leads to stalls...

- **CPU Coupling**
  - DMA channels can decouple the CPU from I/O
  - memory-mapped I/O strongly couples CPU to I/O (devices need more attention)

Memory Hierarchy (from least to most latency)

- Level-1 Cache
- Single cycle access
- Harvard Architecture (separates data and instructions)
- Level-2 Cache or TCM
- MMR (memory mapped registers)
- Main Memory - SRAM, SDRAM, DDR
- MMIO
- nonvolatile memory: Flash, EEPROM, NOVSRAM

→ the way your memory is organized and how your devices are "memory mapped" can affect latency and have a substantial impact for your real-time system performance.