What is a
Real Time Operating System?

Review

We tentatively defined an Operating System to be:

“Some sort of software that you did not write yourself that allows you to use a processor effectively.”

What does Wikipedia say?

- A **real-time operating system** (**RTOS**) is an operating system (OS) intended to serve real-time application process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.
- A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is *jitter*. A *hard* real-time operating system has less jitter than a *soft* real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or *generally* meet a *deadline* is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS.

Dissecting the definition

- An RTOS is useful for:
  - Processing data ‘as it comes in’.
  - Processing data without a lot of delay (buffering).
  - Processing data with delays under 1/10th of a second.
  - Is consistent in the time it takes it to do something (minimize jitter).
  - May be considered hard or soft based on the amount of jitter.
  - Has nothing to do with the amount of data being processed.

Adding to the definition

- An RTOS is useful for:
  - Processing data ‘as it comes in’.
  - Processing data without a lot of delay (buffering).
  - Processing data with delays under 1/10th of a second.
  - Is consistent in the time it takes it to do something (minimize jitter).
  - May be considered hard or soft based on the jitter.
  - Has nothing to do with the amount of data being processed.
- Supports multiple threads of execution (tasks)
- Provides the necessary tools to support multiple tasks.
- Allows the user to schedule and prioritize tasks without surprises.

Definition of an RTOS

“Some sort of software that you did not write yourself that allows you to use your processor effectively to crunch data as fast as it comes in using multiple threads of execution.”

Alternate Definition of an RTOS

“It’s just a multi threaded (MT) OS on a diet.”

A lot of what I will be talking about is the same for an MT OS or an RTOS!

Real Time?

- Real Time refers to being able to keep up with data as it happens in the real world.
- A model of a nuclear bomb explosion may take weeks to simulate what happens in the first 1/10th of a second. This is not real time.
- Processing wheel data for an anti lock braking system (ABS) is real time. Would be worthless if it told you to brake 2 minutes ago!
- This is the “as fast as it comes in” part of the definition.

RTOS vs MT OS

- Both can run multiple tasks at the same time.
- Both have mechanisms for managing tasks.
- Both can crunch data at alarming rate.
  - Based on CPU speed and memory
- So Windows and Linux are RTOS's?

Windows/Linux are not RTOS’s

- An RTOS is
  - Much “lighter weight”
    - No Solitaire, or “Clippy”
    - Provides an embedded solution not for general applications.
  - No hidden tasks
    - In an RTOS you are aware of EVERY task that runs.
      - Ever wonder what the 121 processes in Task Manager do?
  - Guaranteed maximum latency jitter
  - Normally the RTOS is compiled into your application.
    - Notable exception is VxWorks.

What is Latency?

- Latency is the time between an event and the time the task starts to process it.
  - Byte comes in on serial port and generates an interrupt. How long before we enter into the ISR?
  - The ISR then reads the byte from the UART and puts in into a queue. How long before the task waiting on the queue starts to run?
- Note that the RTOS only guarantees a minimum variability in latency (jitter), not the latency itself. (The real latency is a function of processor type, speed, and scheduler frequency.)

Why might I want an RTOS?

- Computer theory says anything you can do with and RTOS you can do in ‘normal’ single task programming.
- It just makes some things soooooooo much easier to accomplish.
- Discard the overhead of a normal OS. Need less memory and processor power.
- You are writing an embedded application that needs to operate without slowing down.
  - How would you like your ABS computer to be running Windows?

Examples of RTOS

- FreeTOS (available for Arduino)
- Micrium (µC/OS-II or III) (mostly ARM)
- VxWorks (Big daddy in the RTOS world)
- ucLinux (paired down Linux)
- PowerPack/embOS (IAR compiler for ARM)
- QNX
- …

Basic principles

1. Allows a single processor to run multiple programs (tasks/threads) at the same time.
2. Allows application to control the priority of each task.
3. Allow tasks to share data safely.
4. Allow tasks to signal each other.

These are the same for any MT OS.

How do tasks work?

- The OS has a Scheduler.
- Scheduler maintains a list of tasks and what line of code they last ran.
- When scheduler gets called, it picks the task with the highest priority that is ready to run.
- It magically restores the processor back to the line of code that the task had been running.

How does that magic happen?

- Each task must have its own pool of memory that the scheduler uses to keep track of it.
- The pool includes:
  - Registers
  - MMU settings
  - FPU settings
  - Priority
  - Run state (running/paused)
  - Things it might be waiting for.
  - Pointer to the task’s stack.
- The pool is often called a Task Control Block (TCB)

How does that magic happen?

- Because the Scheduler is running as an interrupt, the state of the previous task is available on the stack.
- Saves the state of the processor just prior to the interrupt to the current task's TCB.
- Restores the next task's state from its TCB.
- Restores the next task's instruction pointer
  - And hence the next instruction executed will be for the new task
- The scheduler is nasty assembler code.

So how can a task not be ready?

- A task is not ready if:
  - If it has yet to be started.
  - A task requested it to be paused/suspended.
  - A task is waiting/sleeping for some time.
  - A task is waiting for a resource to be free.
  - A task is waiting for another task to give it something.

What happens if no Tasks are ready?

- Introducing the Idle task.
- The idle task is just an infinite loop.
- Sometimes the processor can be put into low power mode in the Idle task.
- A working simple Idle task is often provided by the OS.

Reevaluating the schedule

- The Scheduler is not just called from the timer interrupt.
- When any RTOS system call is made, the Scheduler is often called.
- If a task calls the ‘sleep’ routine, there is no reason to keep running this task.
  - Inside the ‘sleep’ routine, the scheduler is called putting this task in the ‘not ready till later’ list.
  - The next highest priority task that is ready to run gets swapped back in.

Types of Schedulers

- Cooperative
  - No time based scheduler. All tasks must actively call the scheduler at times. Very dated.
- Round Robin
  - All tasks get an equal share of time regardless of priority.
- Priority
  - Highest priority task gets run till such time as it actively yields the processor.
- Preemptive (Hallmark of a good RTOS)
  - Scheduler is reevaluated each time slice as well as at system events.
  - Scheduler normally runs periodically at n Hz based on an interrupt from a HW timer.

Example 1

Daryl’s Laser Range Finder room scanner.

```
void setup(void) {
  stuff
}

void loop(void) {
  for (h=-180, h<180, h++) {
    setHorizontalServo(h);
    for (v=-180, v<180, v++) {
      setVerticalServo(v);
      range = TakeLRFSample();
      SendDataToPC(h, v, range);
    }
  }
}
```

Example 1a

```
Task 1: Servo scanner
Do {
 for (h=-180, h<180, h++) {
  setHorizontalServo(h);
  for (v=-180, v<180, v++) {
   setVerticalServo(v);
   wakeupTaks2();
  }
 }
}
```

```
Task2: RangeFinder

Do {
 range = TakeLRFSample();
 SendDataToPC(range);
 SuspendMySelf();
 /* Will pop back to life here */
}
```

```
Application:
  Setup RTOS stuff
  Create Task1
  Create Task2
  Start RTOS
```

Example 1b

Task 1: Servo scanner

```
Do {
  for (h=-180, h<180, h++) {
    setHorizontalServo(h);
    for (v=-180, v<180, v++) {
      setVerticalServo(v);
      postEventToQueue();
    }
  }
}
```

Task2: RangeFinder

```
Do {
  WaitForEventInQueue();
  range = TakeLRFSample();
  SendDataToPC(range);
}
```

Application:
  Setup RTOS stuff
  Create Task1
  Create Task2
  Start RTOS

Example 1c

Task 1: Servo scanner

```
Do {
  for (h=-180, h<180, h++) {
    setHorizontalServo(h);
    for (v=-180, v<180, v++) {
      setVerticalServo(v);
      postEventToQueue(h,v);
    }
  }
}
```

Task2: RangeFinder

```
Do {
  WaitForEventInQueue();
  h,v = PullItemFromQueue();
  range = TakeLRFSample();
  SendDataToPC(h, v, range);
}
```

Application:

```
  Setup RTOS stuff
  Create Task1
  Create Task2
  Start RTOS
```

Example 1d

```
Task 1: Servo scanner
Do {
 for (h=-180, h<180, h++) {
  setHorizontalServo(h);
  for (v=-180, v<180, v++) {
   setVerticalServo(v);
   postEventToQueueA(h,v);
   WaitForEventInQueueB();
  }
 }
}
```

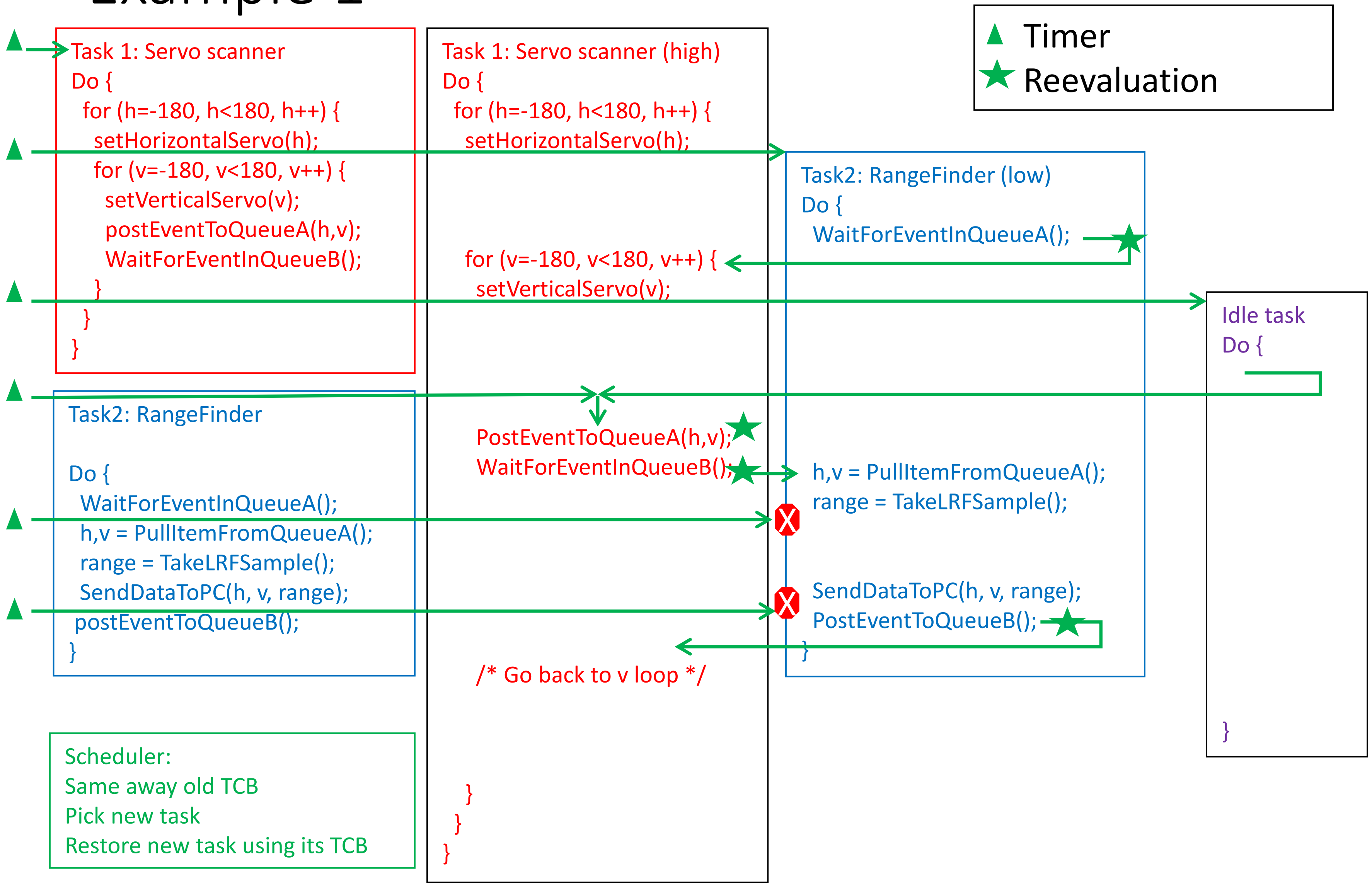
```
Task2: RangeFinder

Do {
 WaitForEventInQueueA();
 h,v = PullItemFromQueueA();
 range = TakeLRFSample();
 SendDataToPC(h, v, range);
postEventToQueueB();
}
```

Application:
- Setup RTOS stuff
- Create Task1
- Create Task2
- Start RTOS

Time

Example 1



▲ Timer
★ Reevaluation

```
Task 1: Servo scanner
Do {
  for (h=-180, h<180, h++) {
    setHorizontalServo(h);
    for (v=-180, v<180, v++) {
      setVerticalServo(v);
      postEventToQueueA(h,v);
      WaitForEventInQueueB();
    }
  }
}
```

```
Task2: RangeFinder

Do {
  WaitForEventInQueueA();
  h,v = PullItemFromQueueA();
  range = TakeLRFSample();
  SendDataToPC(h, v, range);
 postEventToQueueB();
}
```

```
Scheduler:
Same away old TCB
Pick new task
Restore new task using its TCB
```

```
Task 1: Servo scanner (high)
Do {
  for (h=-180, h<180, h++) {
    setHorizontalServo(h);

    for (v=-180, v<180, v++) {
      setVerticalServo(v);

      PostEventToQueueA(h,v);
      WaitForEventInQueueB();

      /* Go back to v loop */

    }
  }
}
```

```
Task2: RangeFinder (low)
Do {
  WaitForEventInQueueA();

  h,v = PullItemFromQueueA();
  range = TakeLRFSample();

  SendDataToPC(h, v, range);
  PostEventToQueueB();
}
```

```
Idle task
Do {

}
```

Seems overly complicated!

- For this simple application you would be right.
- But what if the LIDAR scanner was on a moving robot with other sensors?
  - How would you receive data from other sensors, plot your course, control the drive motors, and still scan the LIDAR?
- Or how about a system with a bunch of sensors all running at different rates?
  - I have a 58 thread application at work.
  - My AVC robots has about 10

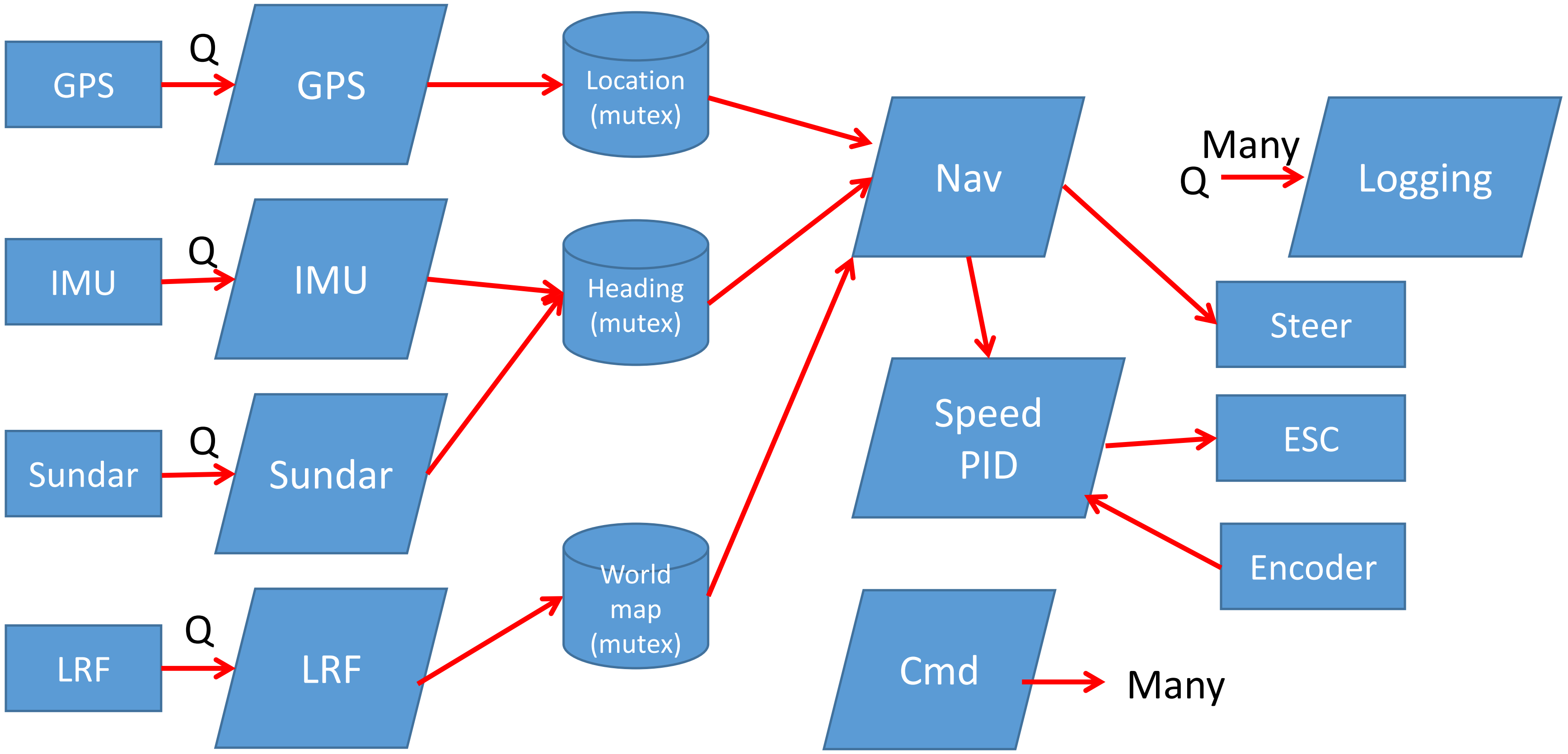
Example 2 (AVC robot)

- Task1 – Navigation (5 Hz)
- Task2 – Speed PID (20 Hz)
- Task3 – SD card logging (as needed)
- Task4 – GPS (as needed ~5 Hz)
- Task5 – IMU (200 Hz)
- Task6 – Command processor (as needed)
- Task7 – LRF (5 Hz)
- Task8 – Solar Compass (5 Hz)
- Task9 – Temp/voltage monitor
- …

Example 2 (Ugly mess)

```
Do {
 if (time to try to read data from GPS) {
  Read data, process it;
 }
 if (bytes available from the command parser) {
  Read bytes, try to parse and execute them
 }
 if (time to update the speed PID) {
  Get current speed, perform computations, set ESC
 }
 if (time to perform navigation) {
  Perform math; Set steering, set desired speed;
 }
 …
}
```

Example 2 (MT solution)



Example 2 (detail)

So why did SD card logging get its own task?

- I did not want the possibility that a slow SD card would back up the system.
- To log data, I put the data in a fixed sized queue.
- If the queue is full, that data gets dropped.
    - Oh well…
- These types of things are where an RTOS rocks.

RTOS toolbox

- Task control
- Mutex
- Semaphore
- Queue
- Timers
- Memory allocation

Task Control

- Start and stop tasks
- Pause and resume a task
- Sleep
- Change a task’s priority
- Force a scheduler reevaluation
- Determine what task I am
    - In case you have multiple identical tasks

Example Task Creation

“EmbOs”

```
static  uint32_t MyTaskStack[512];  /* Stack for the task */

/* Create the task */
RTOS_CreateTask(tcbMyTask,                          /* Task Control Block */
                 “MyTaskName",                      /* Task Name */
                 MyTask,                            /* Task routine */
                 OS_TASK_PRIORITY_MY_TASK,                  /* Priority */
                 MyTaskStack );                     /* Stack */


void MyTask(void) {
  while (1) {
    /* Do stuff here */
  }
}
```

Mutex

The dirty secret of multiple thread programming is resource sharing.

Suppose the GPS task is part way through updating the position (lat and lon) when the scheduler swaps it out for the navigation task.

And what happens if the navigation task now uses the lat/lon in a computation?

Well the navigation will use the current latitude but the previous longitude.

Mutex Example

```
GpsPosition_t myLocation;  /* The current location of the robot. Structure of lat/lon */

void setGpsPosition(void)  {
  GpsPosition_t Pos;
  Pos = getPositionFromGps();
  OS_TakeMutex(positionMutex);
  {
    myLocation.lat = pos.lat;
    myLocation.lon = pos.lon;
  }
  OS_GiveMutex(positionMutex);
}

GpsPosition_t getGpsPosition(void) {
  GpsPosition_t retval;
  OS_TakeMutex(positionMutex);
  {
    retval = myLocation;
  }
  OS_GiveMutex(positionMutex);
  return retval;
}
```

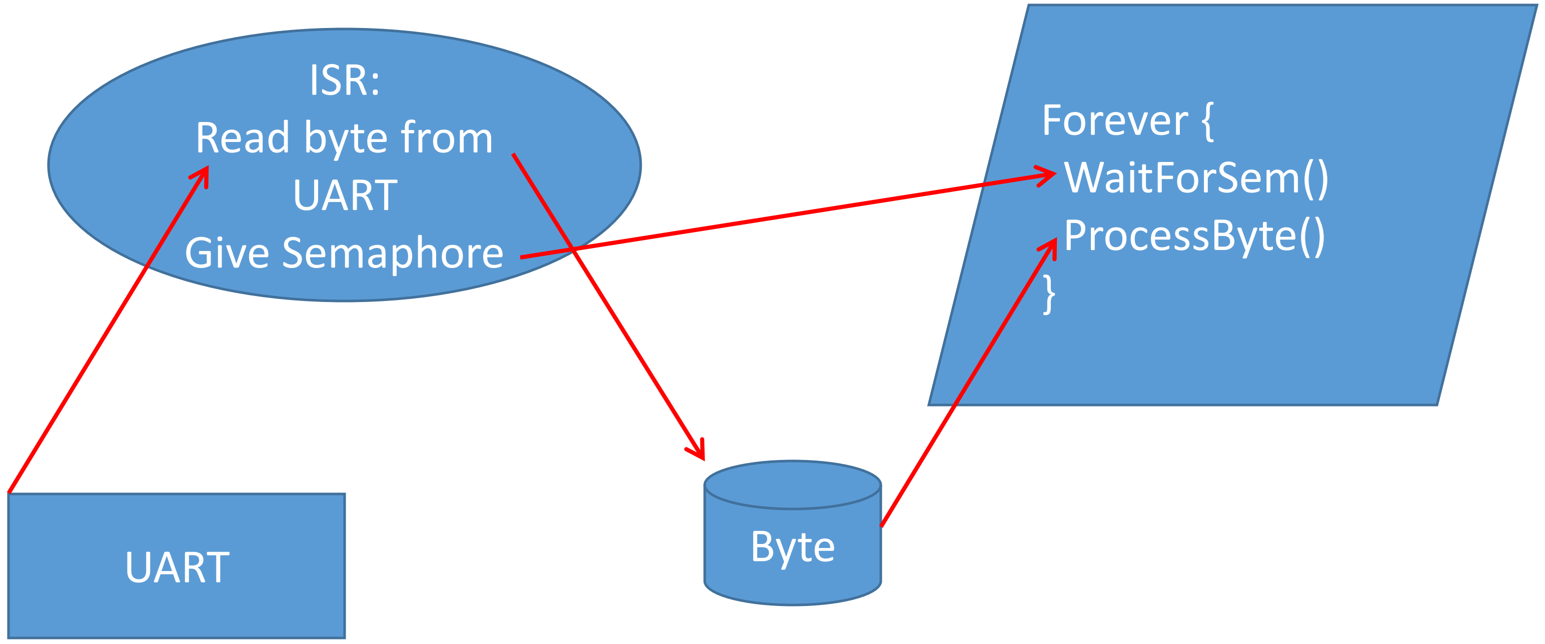
Mutex Operations

- Take and Release
- Take with timeout
- Query state
- Causes a scheduler reevaluation

Semaphore

- Very much like a Mutex.
- Where only the task that takes the Mutex can release it, a semaphore can be taken by one thread and released by another.
- Semaphores are useful to signal events from one task to another.
  - ISR may give a semaphore to indicate a byte was received. A task can be pending on that semaphore to wake up and process the byte.

Semaphore


ISR:
Read byte from UART
Give Semaphore
Forever {
WaitForSem()
ProcessByte()
}
UART
Byte

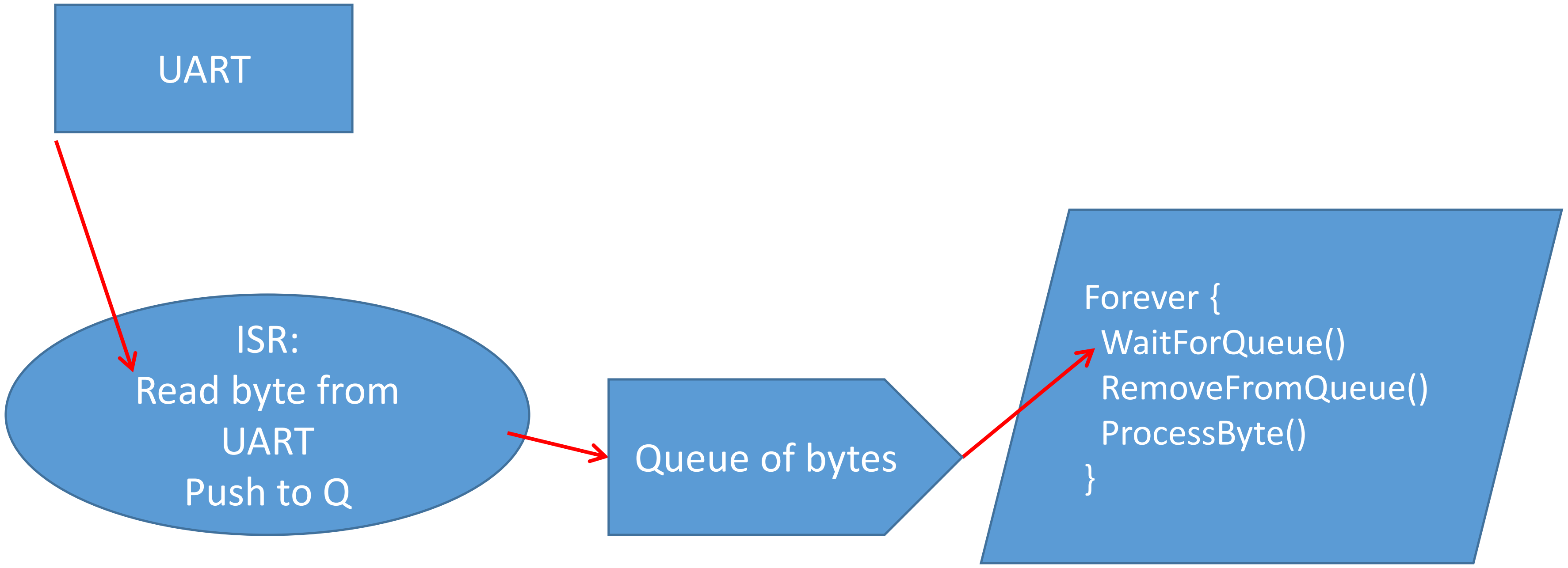
Semaphore Operations

- Take and Release
- Take with timeout
- Query state
- Causes a scheduler reevaluation

Queue

- A way to send data from one place to another.
- Queues are normally fixed in size.
- If you try to add to a queue that is full, you will either wait till there is room, or you can get an error.
- Items in the queue can be single bytes or chunks of data like a C structure.
- Queue is protected so that only one task can modify it at a time.
- Causes a scheduler reevaluation

Queue



```
Forever {
  WaitForQueue()
  RemoveFromQueue()
  ProcessByte()
}
```

Simpler and safer than the Semaphore example!

May not be possible if your RTOS does not have a queue you can push to from an ISR. But you can always roll your own!

Queue Operations

- Create/Destroy
- Put/Get
- Get with timeout
- Query state
    - How many items in queue?
    - How big is the queue?
    - Anyone waiting on the queue?
- Flush

Timers

- Schedule some routine to get called in the future.

Timers

```
…
Ok = getGpsLocation()
If (Ok) {
  clearTimer(NoGpsTimer);
  setTimer(NoGpsTimer, 2, cbNoGps);
}
…


Void cbNoGps(void)  {
  /* Crap, we failed to get GPS data, panic stop */
  MotorToFullStop();
  AbortProgram();
}
```

Timer Operations

- Create/delete
- Single shot start.
  - Timeout in 1 second and call my routine
- Repetitive start.
  - Timeout every 500ms and call my routine
- Stop
  - Stop timer and don’t call my routine
- Query
  - Is timer running?

Memory Allocation

- Allocate and deallocate chunks of memory.
  - Many embedded purists would say you should NEVER dynamically allocate memory.
  - Most people would say you should only ever use these functions once during initialization.
  - Those that use them during execution may find that memory segmentation at some point prevents you from getting the memory you want.

Questions?