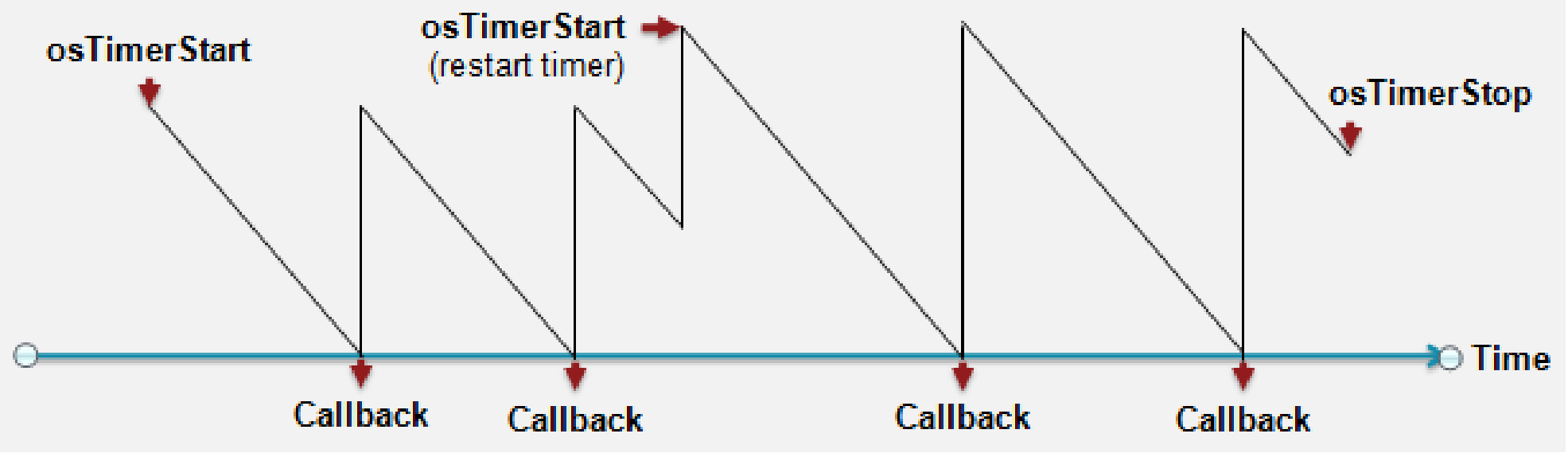
Advanced RTOS Issues: Virtual Timers, More Thread Features, RAM Requirements and Debugging

Virtual Timer Objects



- Runs a ***callback function*** when timer expires
  - ISR or osRtxTimerThread may call the function
- What else?
  - Type is osTimerId_t
  - Timer counts scheduler ticks (e.g. 1 ms period)
  - Timer can be one-shot or periodic
  - Timer can be started, stopped, restarted
- API: osTimer[New | Delete | Start | Stop | IsRunning | GetName]
- https://arm-software.github.io/CMSIS_5/RTOS2/html/group__CMSIS__RTOS__TimerMgmt.html#details

Advanced Thread Functions

- osThreadYield
  - Thread suspends self, yields control to another thread at same priority level (but not lower). If none, then continue.
- osThreadSuspend(tid)
  - Move thread *tid* to blocking state
- osThreadResume(tid)
  - Move suspended thread *tid* to ready state
- osThreadExit
  - Thread terminates self
- osThreadTerminate(tid)
  - Terminate thread *tid*

- Stack
  - osThreadGetStackSize: how much space in total was the stack allocated?
  - osThreadGetStackSpace: how much stack space is free?
- Others
  - osThreadDetach
  - osThreadJoin

Custom Attributes for Threads

struct osThreadAttr_t

Data Structure Documentation
Specifies the following attributes for the osThreadNew function.

Data Fields		
const char *	name	name of the thread. Pointer to a constant string with a human readable name (displayed during debugging) of the thread object. **Default**: *NULL* no name specified (debugger may display function name instead).
uint32_t	attr_bits	attribute bits. The following bit masks can be used to set options: •**osThreadDetached** : create thread in a detached mode (default). •**osThreadJoinable** : create thread in **joinable mode**.
void *	cb_mem	memory for control block. Pointer to a memory for the thread control block object. Refer to **Static Object Memory** for more information. **Default**: *NULL* to use **Automatic Dynamic Allocation** for the thread control block.
uint32_t	cb_size	size of provided memory for control block. Size (in bytes) of memory block passed with **cb_mem**. For RTX, the minimum value is defined with **osRtxThreadCbSize** (higher values are permitted). **Default**: *0* as the default is no memory provided with **cb_mem**.
void *	stack_mem	memory for stack. Pointer to a memory location for the thread stack (64-bit aligned). **Default**: *NULL* to allocate stack from a fixed-size memory pool using **Thread Stack Management**.
uint32_t	stack_size	size of stack. The size (in bytes) of the stack specified by **stack_mem**. **Default**: *0* as the default is no memory provided with **stack_mem**.
osPriority_t	priority	initial thread priority (default: osPriorityNormal) Specifies the initial thread priority with a value from **osPriority_t**. **Default**: *osPriorityNormal*.
TZ_ModuleId_t	tz_module	TrustZone module identifier. TrustZone Thread Context Management Identifier to allocate context memory for threads. **Default:** *0* not thread context specified.
uint32_t	reserved	reserved (must be 0) Reserved for future use.

- https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__ThreadMgmt.html#structosThreadAttr__t

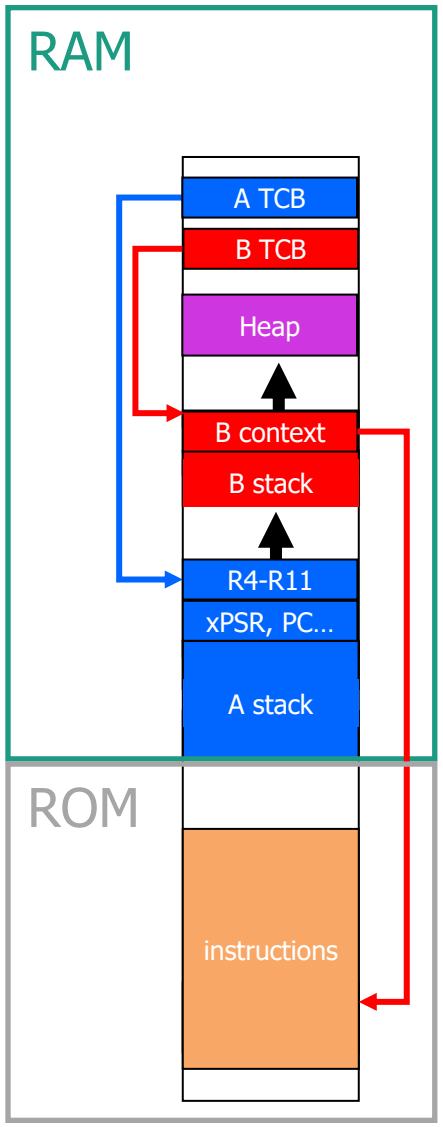
Creating Objects with Custom Attributes

```
const osThreadAttr_t Read_TS_attr = {
    .priority = osPriorityNormal
};
const osThreadAttr_t Read_Accelerometer_attr = {
    .priority = osPriorityBelowNormal,
    .stack_size = READ_ACCEL_STK_SZ
};
const osMutexAttr_t LCD_mutex_attr = {
     .name = "LCD_mutex",      // human readable mutex name
     .priority = osMutexPrioInherit    // attr_bits
};
void Create_OS_Objects(void) {
    LCD_mutex = osMutexNew(&LCD_mutex_attr);
    t_Read_TS = osThreadNew(Thread_Read_TS, NULL, &Read_TS_attr);
    t_Read_Accelerometer = osThreadNew(Thread_Read_Accelerometer, NULL,
                                       &Read_Accelerometer_attr);
}
```

ANALYSIS OF RAM REQUIREMENTS

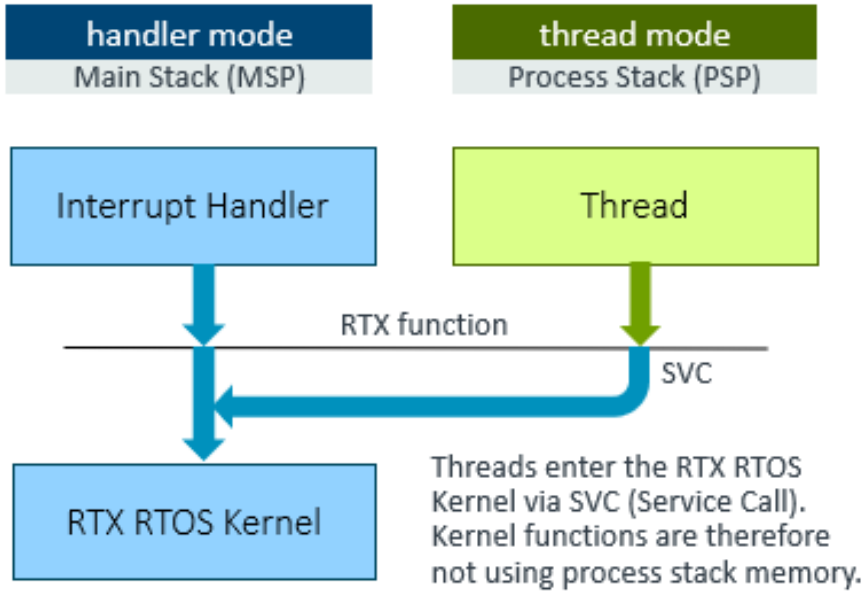
How Much RAM Do We Need?

- Function call stacks
- OS objects (including control blocks)
- Heap memory (may include OS objects)



How Much RAM for Stacks?

- Documentation
- How many stacks? One for OS, one per thread
- How large?
  - Space for stacked thread context
    - No floating point unit? 64 bytes
    - Floating point unit? 200 bytes
  - Space for each thread's deepest calling behavior
    - Include registers stacked by hardware. Handlers use MSP, not PSP
    - Main stack size for RTX v5 documented here
    - Exact analysis is hard for a non-trivial program
    - How to estimate?
      - Analyze anyhow – usually overestimates. Linker provides stack depth analysis
      - Test and measure – usually underestimates. RTOS provides stack usage watermark and stack overflow detection.
- Covered in ECE 461/561 in detail



How Much RAM for RTOS Objects?

Category	Control Block Size Attribute	Size	#define symbol
Thread Management	osThreadAttr_t::cb_mem	68 bytes	**osRtxThreadCbSize**
Timer Management	osTimerAttr_t::cb_mem	32 bytes	**osRtxTimerCbSize**
Event Flags	osEventFlagsAttr_t::cb_mem	16 bytes	**osRtxEventFlagsCbSize**
Mutex Management	osMutexAttr_t::cb_mem	28 bytes	**osRtxMutexCbSize**
Semaphores	osSemaphoreAttr_t::cb_mem	16 bytes	**osRtxSemaphoreCbSize**
Memory Pool	osMemoryPoolAttr_t::cb_mem	36 bytes	**osRtxMemoryPoolCbSize**
Message Queue	osMessageQueueAttr_t::cb_mem	52 bytes	**osRtxMessageQueueCbSize**

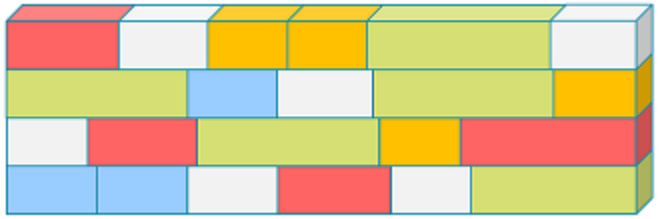
- Objects only created at program start-up, never deleted?
  - Yes? Relatively easy
  - No? Much harder. Objects are dynamically allocated (…new, …delete) as program runs. Determine worst-case memory requirements
- Remember to include additional RAM for memory pools, message queues
- Defined in rtx_os.h, see https://arm-software.github.io/CMSIS_5/RTOS2/html/pControlBlockSizes.html

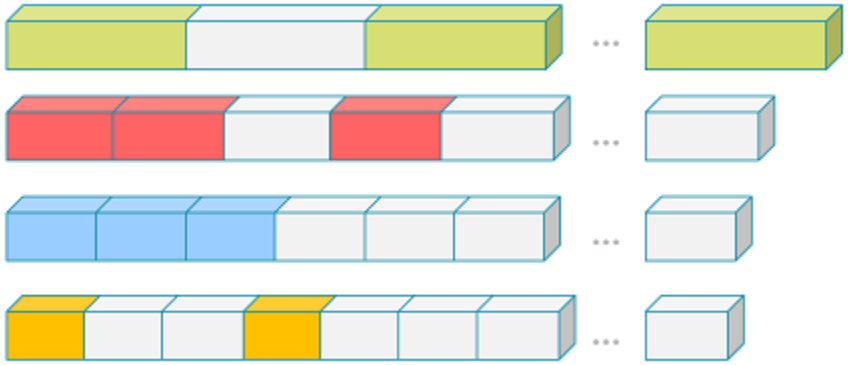
How Should We Split the RAM?

- Limited amount of RAM
- Hard to know exact size needed for
  - Thread stacks
  - Heap
  - Dynamic OS objects (message queues, memory pools)

Memory Allocation Models

- Three memory allocation models (documentation)
  - Global Dynamic (default): One pool for all objects, uses heap
  - Object-specific Dynamic: One pool per memory type
  - Static: Allocate at compile time
- Can use all types in one program based on needs





Option	Value
System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	☑
ISR FIFO Queue	16 entries
Object Memory usage counters	☐

Option	Value
Semaphore Configuration	
Object specific Memory allocation	☐
Number of Semaphore objects	1

Option	Value
Message Queue Configuration	
Object specific Memory allocation	☐
Number of Message Queue objects	1
Data Storage Memory size [bytes]	0

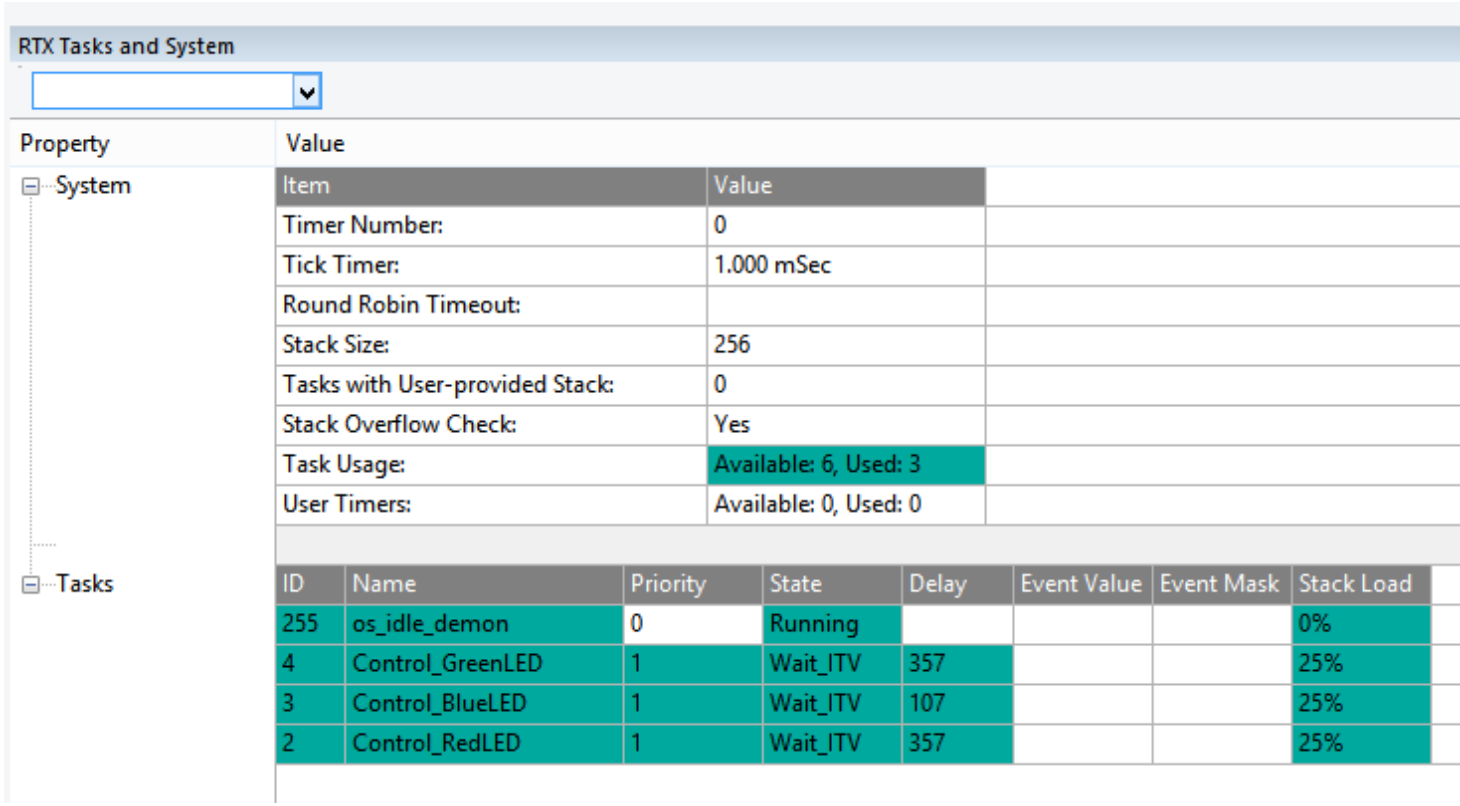
When to Use the Memory Allocation Models

Model	Pros	Cons
Global Dynamic	Simple, no external fragmentation	Creating and destroying objects of different sizes can cause internal fragmentation
Object-Specific Dynamic	Deterministic, no internal fragmentation	Requires some analysis and multiple pools, can suffer from external fragmentation
Static	Deterministic, prevents all fragmentation	Requires detailed analysis for tight bounds (safe but not wasteful)

DEBUGGING

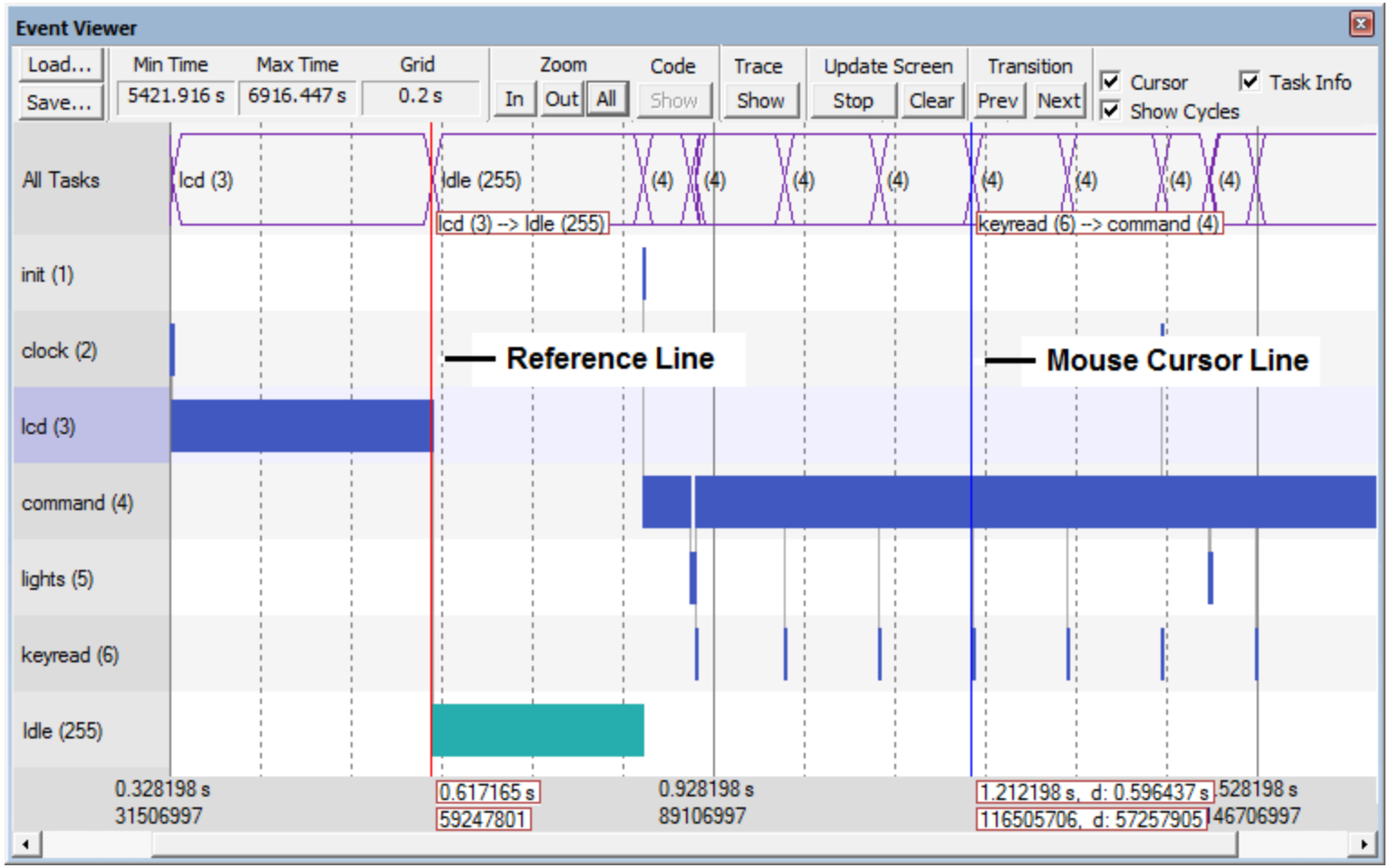
RTX & MDK Debugging Support

- Debugging multithreaded system is challenging
  - Thread interactions
  - Multiple states
- MDK has integrated RTX debug support
  - Improves visibility into system state
  - RTX OS status window available in MDK
    - Debug -> OS Support -> RTX threads and System
  - Displays …
    - What thread is doing
    - What thread is waiting for
    - Other data



Event Viewer

- Shows thread-oriented timeline view of system operation
  - Which thread executes when
  - Context switches
  - Interrupt service routines
- Not available on KL25Z MCU (no SWV support)



Common Issues and Bugs

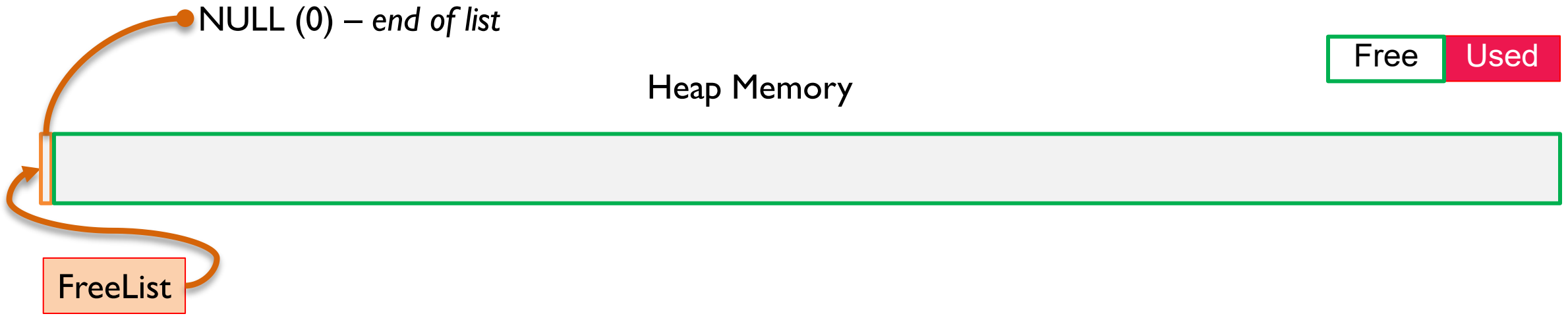
- Crash: Does each thread have enough stack space?
  - sprintf takes a lot. Floating point math does too.
- Did you scale the RTOS to your application?
  - Number of threads, semaphores, queues, mailboxes, etc.
- Always check result/error codes for system calls
  - Light an LED if there's an error, possibly put an infinite loop there
- Why doesn't my thread run?
  - Look at scheduler's data structures via debugger
    - Threads which are ready to run
    - TCB: thread control block
  - If the error LED goes on, set a breakpoint there and see what happened

More Common Issues and Bugs

- Is your thread structured as an infinite loop with an OS call on which to block?
- Are interrupts working properly? Try substituting a polling function to isolate the problem.
- Is there enough memory for your program? Check the .map file
- Don't use printf or write to an LCD in time-critical code (too slow)
- Put as little code into ISRs as possible
- Be careful about which functions and API calls you use in ISRs

OLD

Example Operations



- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B. Update list entry with (size of B)-N
  - Update list pointers as needed
  - Return pointer to start of allocated block
- Deallocation of block (free)
  - If block to free is adjacent to a block in FreeList, merge them and update the list entry
  - Else add list entry with pointer to B (and size) to FreeList

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
c = (int *) malloc(64);
```