Real Time Operating Systems

LY ETRX
Sem VIII

Arati Phadke
AY 2021-22

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya
TRUST

Computer Organization and Architecture

Course Outcomes

At the end of successful completion of the course the student will be able to

CO1: Understand real time operating system concepts.

CO2: Create various instances of a task and send and receive data to and from a queue

CO3: Implement the APIs within an Interrupt Service Routine

CO4: Develop algorithms for real time processes using FreeRTOS

CO5: Understand memory allocation schemes

Module 5

Memory Management

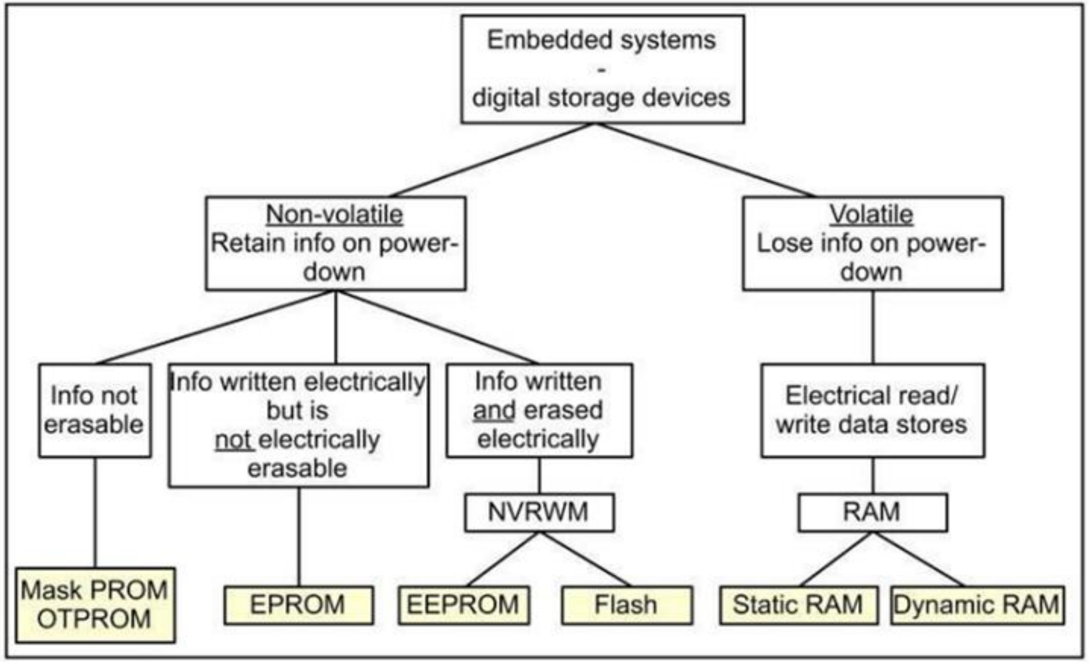
5.1 Memory allocation schemes, Heap related utility functions

Memory concepts

Virtual Memory, Physical memory and Cache Memory

- Size requirements
- Speed Requirements
- Development of processors handling Virtual Memory, paging
- Role of Memory Management Unit of OS

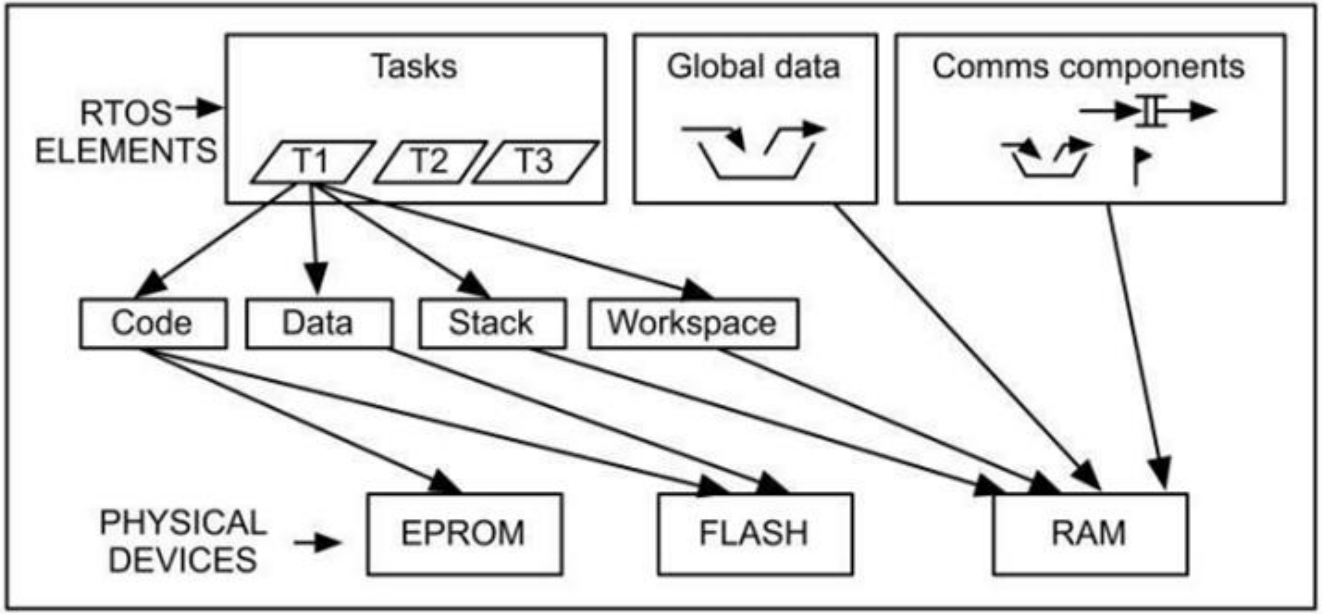
Memory


Embedded systems
-
digital storage devices
Non-volatile
Retain info on power-down
Volatile
Lose info on power-down
Info not erasable
Info written electrically but is not electrically erasable
Info written and erased electrically
Electrical read/ write data stores
NVRWM
RAM
Mask PROM
OTPROM
EPROM
EEPROM
Flash
Static RAM
Dynamic RAM

Embedded systems - memory device organization

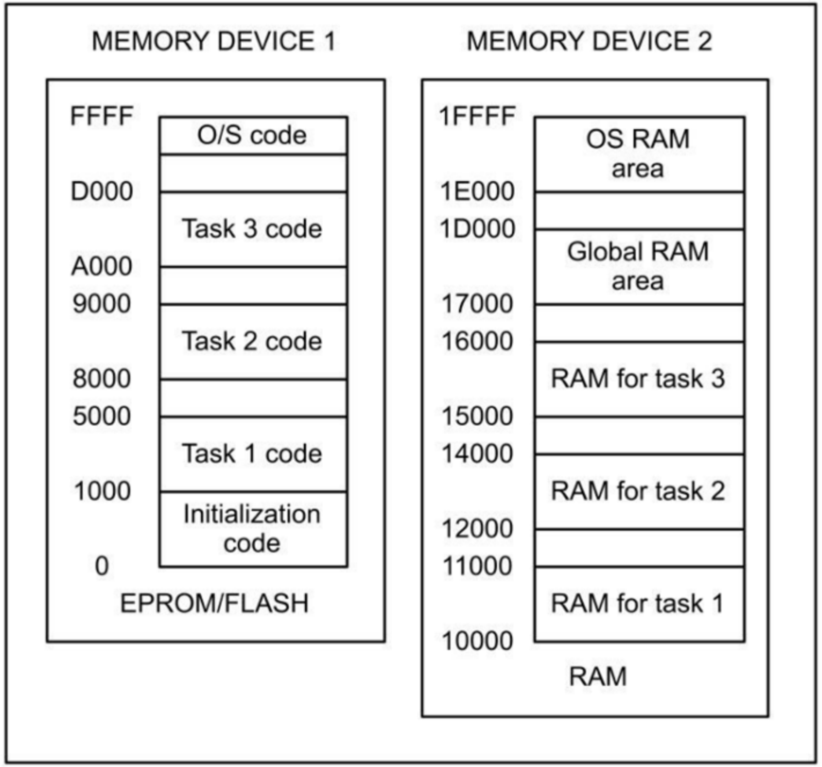
- code and essential data (both constants and variables) : in non-volatile storage.
- **Variable data?**
- **Weather station measurements : RAM.**
- **Car radio station allocation using selection buttons : Flash**
- **State of charge of a submarine propulsion battery. :** need for nonvolatile storage of data. But, the number of settings and chances of rewrite are very low for radio, but for battery management the number of rewrites may wear out flash memory…so ongoing data to RAM and power down transfer RAM to Flash
- **Use of cache memory**

Memory aspects - conceptual and physical.



Memory aspects - conceptual and physical.

- Multiple tasks handling local memory and shared memory (with virtual processor)
- Local / Private memory houses information specific to a task, including both data and code items
- Shared memory is generally used for global data, communication components and shared code.



MEMORY DEVICE 1

FFFF — O/S code

D000

Task 3 code

A000

9000

Task 2 code

8000

5000

Task 1 code

1000

Initialization code

0

EPROM/FLASH

MEMORY DEVICE 2

1FFFF — OS RAM area

1E000

1D000

Global RAM area

17000

16000

RAM for task 3

15000

14000

RAM for task 2

12000

11000

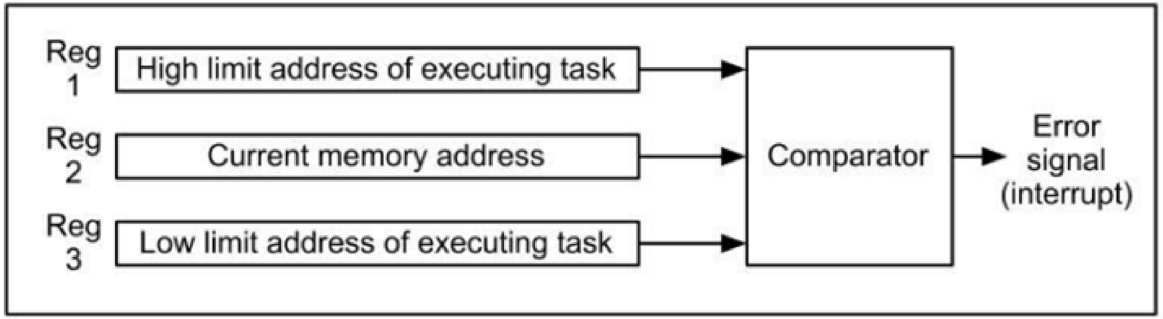
RAM for task 1

10000

RAM

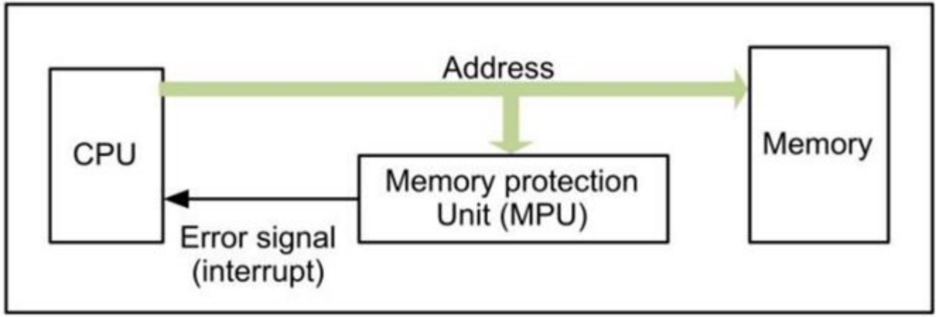
Memory Access Control

- Intertask interference , Interference with OS
- Simple Approach: When a task is dispatched, the memory address bounds are loaded into two registers. Each memory address produced by the processor is compared with these bounds; any violation leads to the generation of an error signal (usually an interrupt).
- **Difficulties:** time taken for comparison , getting information of bounds (depends on memory allocation)


Reg 1
High limit address of executing task
Reg 2
Current memory address
Reg 3
Low limit address of executing task
Comparator
Error signal (interrupt)

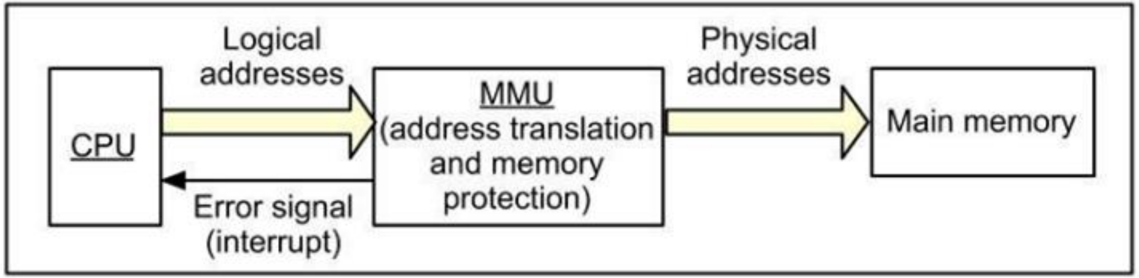
Memory Access Control

- Memory Protection Unit (MPU)
- Memory Protection registers can be useful in multiple tasks
- At task creation time each protection register is configured with the appropriate memory information.
- During task execution each address generated by the processor is compared with those held by its MPR; any attempt to access memory outside of the predefined limits raises an exception.
- many processors now provide an MPU as part of the on-chip circuitry



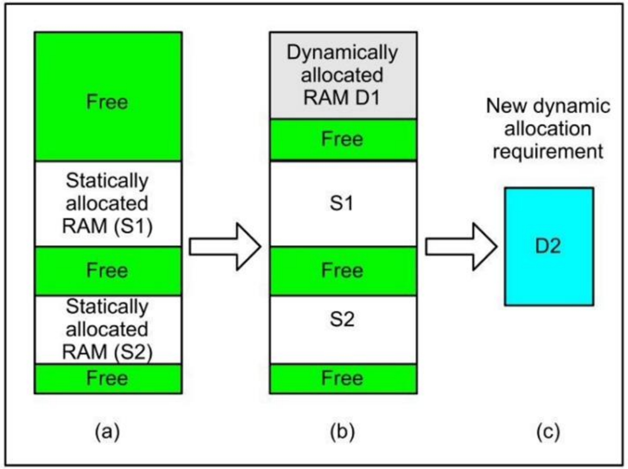
Memory Access Control

- Memory Management Unit (MMU)
- Information about the target memory structure, the physical memory, isn't known at compile time.
- The primary role of the memory management unit (MMU),Logical to Physical Address translation : Set of translation registers
- All operations carried out on and by the MMU (and also the MPU) are 'invisible' to the programmer. Memory bounds, address translation, access rights, etc. are all dealt with by the operating system. Such information is generated during the compilation process, being derived from task creation, linkage and location information



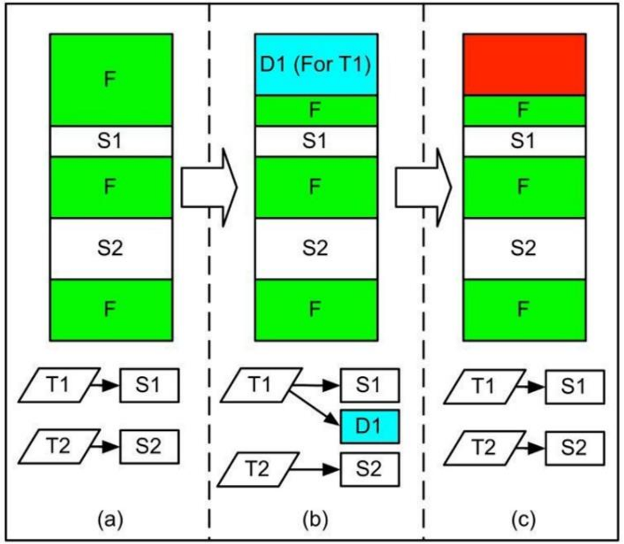
Memory Allocation

- Static Vs Dynamic Allocation
- Problems of Dynamic Memory Allocation
  - Memory allocation and fragmentation
  - Tasks need RAM space: how to provide?
  - At the time of task creation define amount of memory required, explicit allocation left to compiler : Static and remains for lifetime of a task
  - It is wasteful to permanently tie up a resource that is used only intermittently. So Dynamic Allocation
  - if memory allocation and deallocation isn't carefully controlled, the system will almost certainly fail
  - The problem arises because the allocated memory is not nicely, compactly, organised; instead it is fragmented through the available memory space.


Free
Statically allocated RAM (S1)
Free
Statically allocated RAM (S2)
Free
Dynamically allocated RAM D1
Free
S1
Free
S2
Free
New dynamic allocation requirement
D2
(a)
(b)
(c)

Memory Allocation

- Problems of Dynamic Memory Allocation : memory Leakage
- Two tasks T1 , T2 statically allocated memory S1, S2
- Task 1 invokes function which acquires D1 Dynamically (malloc()) and does not deallocate it (free())
- 'leaked' memory from the available pool.
- What if such function is called number of times ? or recursive?
- 'dangling pointers'
- Two techniques: OS activated (implicit) or programmer controlled (explicit

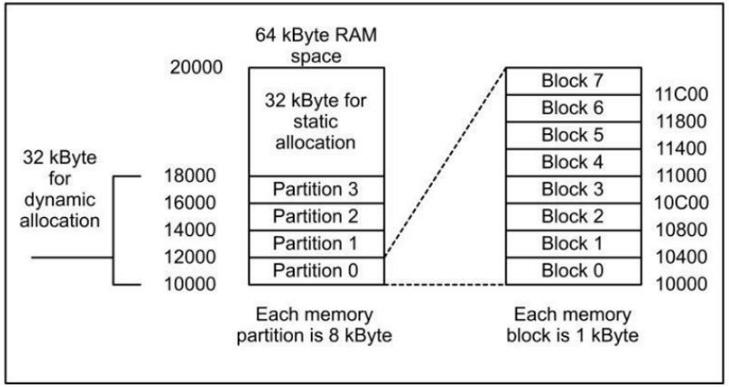


Memory Allocation

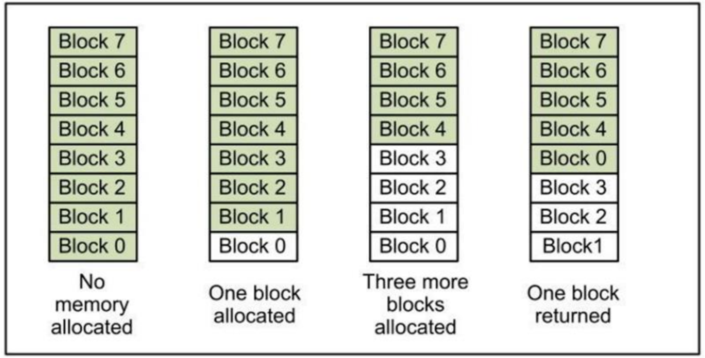
- **OS activated (implicit)**
- When the OS retrieves allocated but unreferenced memory : 'garbage collection'.
- It carries out garbage collection with task called the garbage collector.
- It keeps track of all the memory items (objects) and references to such items. If it sees that an object is no longer referenced, it retrieves the memory space for future use.
- While this method is safe, secure and simple to use, it has one major drawback. The garbage collector task itself introduces uncertainty into the timing performance of the system.
- In designs where the garbage is collected in one go, the effect can be substantial. Such methods are totally unacceptable for hard/fast systems. The problem can be minimized by collecting the garbage incrementally - 'a bit at a time'.
- **Programmer activated (Explicit)**
- One proven, practical technique is to encapsulate matching allocation and deallocation operations within a subprogram. Failing that, check your programs using a run-time analysis tool designed to catch memory leaks.

Secure Memory Allocation

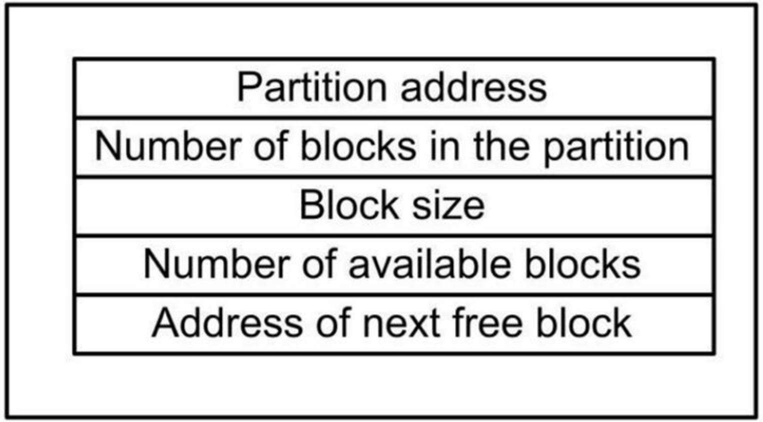
- Available RAM : Divide as static allocation and dynamic Allocation
- Control of allocation and deallocation of memory by an RTOS-provided memory manager
- Memory is allocated from selected partitions. Only one block is allocated for each request made.
- A count for number of blocks currently allocated.
- Deallocated memory is always returned to the partition it came from Only one block is returned for each return request made.
- Allocation/deallocation times are deterministic.
- Errors are flagged up if an allocation request is made to an empty partition or if a return is attempted to a full partition.



Secure Memory Allocation



Block 7	Block 7	Block 7	Block 7
Block 6	Block 6	Block 6	Block 6
Block 5	Block 5	Block 5	Block 5
Block 4	Block 4	Block 4	Block 4
Block 3	Block 3	Block 3	Block 0
Block 2	Block 2	Block 2	Block 3
Block 1	Block 1	Block 1	Block 2
Block 0	Block 0	Block 0	Block1
No memory allocated	One block allocated	Three more blocks allocated	One block returned



Partition address
Number of blocks in the partition
Block size
Number of available blocks
Address of next free block

- Each Partition has memory control block
- Key to the reordering of blocks is the use of address (pointer) techniques to create a linked list of memory locations.

Memory Allocation in FreeRTOS

- Kernel objects such as tasks, queues, semaphores and event groups. are not statically allocated at compile-time, but dynamically allocated at run-time;
- FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is deleted.
- This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.
- What the stack and heap are.
- The standard C library malloc() and free() functions.

malloc() and free()

- Memory can be allocated using the standard C library malloc() and free() functions,
- but they may not be suitable, or appropriate,
  - They are not always available on small embedded systems. Their implementation can be relatively large, taking up valuable code space.
  - They are rarely thread-safe
  - They are not deterministic; the amount of time taken to execute the functions will differ from call to call.
  - They can suffer from fragmentation.
  - They can complicate the linker configuration.
  - They can be the source of difficult to debug errors if the heap space is allowed to grow into memory used by other variables.

Memory Allocation in FreeRTOS

- FreeRTOS treats memory allocation as part of the portable layer (as opposed to part of the core code base).
- Different embedded systems have varying dynamic memory allocation and timing requirements, hence single dynamic memory allocation algorithm is appropriate for a subset of applications.
- Removing dynamic memory allocation from the core code base enables application writer's to provide their own specific implementations, when appropriate.
- When FreeRTOS requires RAM, instead of calling malloc(), use pvPortMalloc().
- When RAM is being freed, instead of calling free(), use vPortFree().
- pvPortMalloc() and vPortFree() are public functions, so can also be called from application code.

Heap Management

- **heap**: part of memory that contains the objects -
  - dynamically grows and shrinks during execution
  - each object is allocated and deallocated independently from the other objects
- Linear Allocation : Next fit
- Block Allocation
  - First-Fit Memory Allocation: first request claims the first available memory with space more than or equal to it’s size.
  - Best-Fit Memory Allocation :smallest to largest.

Dynamic Memory Allocation

Dynamic Memory Allocator

Maintains pool of memory, satisfies requests for memory if possible, receives freed memory back

Has two interfaces : Allocate (size), deallocate(pointer)

Additional interfaces for allocating clear memory, reallocate for expansion

Wild pointer

Dangling pointer

Memory leak

Heap Allocation

Memory Allocation schemes in FreeRTOS

FreeRTOS Stack and Heap Management

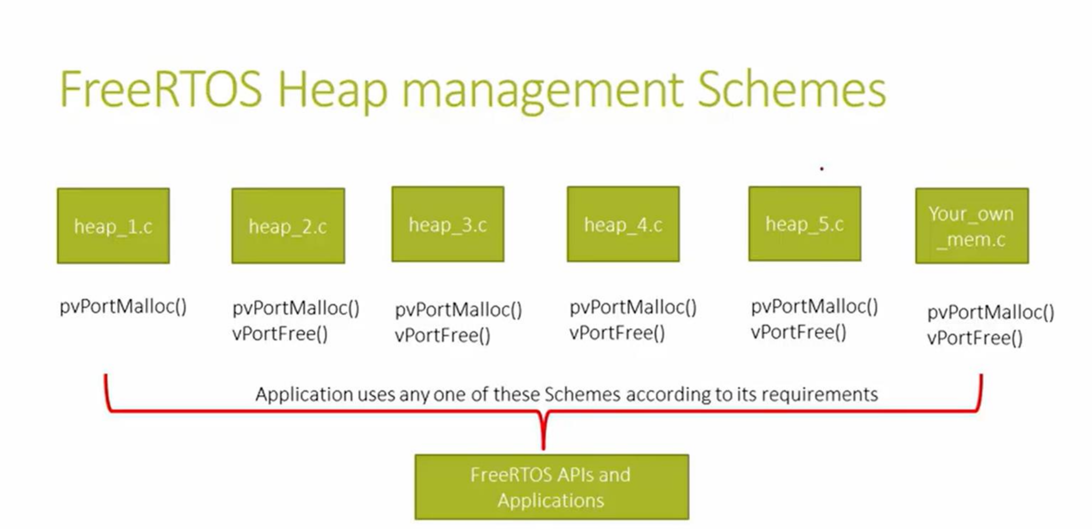
https://www.youtube.com/watch?v=bLQWB4H0TpE

RTOS Memory Management

https://youtu.be/Qske3yZRW5I

Memory Allocation schemes in FreeRTOS

5 different heap management schemes
Heap_1 - pvPortMalloc(),
Heap_2 , Heap_3 , Heap_4 ,Heap_5 - pvPortMalloc(), pvPortFree()



FreeRTOS Heap management Schemes

heap_1.c
pvPortMalloc()

heap_2.c
pvPortMalloc()
vPortFree()

heap_3.c
pvPortMalloc()
vPortFree()

heap_4.c
pvPortMalloc()
vPortFree()

heap_5.c
pvPortMalloc()
vPortFree()

Your_own
_mem.c
pvPortMalloc()
vPortFree()

Application uses any one of these Schemes according to its requirements

FreeRTOS APIs and
Applications

Memory Allocation schemes in FreeRTOS

- heap_1 - Allocation Only : the very simplest, does not permit memory to be freed.
- heap_2 - Best fit without coalescence: permits memory to be freed, but does not coalescence adjacent free blocks.
- heap_3 - Standard library malloc and free: simply wraps the standard malloc() and free() for thread safety.
- heap_4 - First fit with coalescence: coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
- heap_5 - First fit with coalescence over multiple regions : as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas.