Module 2: Memory Management & Queue

Learning Objective:

1. Understand ESP32 Memory Structure
2. Comprehend the Challenges of Memory Management
3. Explore FreeRTOS Memory Management Techniques
4. Implement Dynamic Memory Allocation
5. Handle Memory Issues in Embedded Systems
6. Understand Task Synchronization using Queues

- Module 2: Memory Management & Queue
- Code Sample

Module 2: Memory Management & Queue

Memory Management in ESP32

Memory management is a crucial aspect of developing embedded systems, especially for platforms with limited resources like the ESP32. The ESP32 is a dual-core microcontroller that supports various wireless protocols, such as Wi-Fi, Bluetooth, and BLE. It has limited RAM (520 KB) and flash memory (4 MB), which must be utilized efficiently to run complex applications.

ESP32 manages memory resources using different memory regions and types. The main memory regions include:

- **Internal Memory**: This includes Instruction RAM (IRAM), Data RAM (DRAM), and Read-Only Memory (ROM). IRAM stores executable code, DRAM stores data and heap, and ROM holds boot code and some libraries.
- **External Memory**: This includes external SPI RAM (PSRAM) and flash memory. PSRAM can be used to extend DRAM, while flash memory can store application code and data through memory mapping.

The main types of memory are:

- **Static Memory**: Allocated at compile-time, with fixed size and location. This includes global and static variables, constants, and literals.
- **Dynamic Memory**: Allocated at runtime with flexible size and location. This includes local variables, heap, and stack.

One challenge with efficient memory use on the ESP32 is avoiding **memory fragmentation**, which occurs when available memory is split into small, non-contiguous blocks that cannot be used for allocation requests. Memory fragmentation can reduce system performance and stability, and even lead to memory allocation failures.

Another challenge is ensuring **memory alignment** with the CPU architecture's word size. The ESP32 uses a 32-bit architecture, meaning each word is 4 bytes. If memory access is not aligned with word boundaries, it can increase CPU cycles and cause bus errors.

FreeRTOS, the real-time operating system kernel, aids memory management on the ESP32 by providing:

- **Multitasking scheduling**, which allocates CPU time to tasks based on their priority and status. Each task has its own stack, which grows and shrinks dynamically based on the task's needs.
- **Flexible memory allocation options**, allowing developers to choose the most appropriate scheme for their applications. FreeRTOS provides five different heap implementations with varying complexity and features.
- **Task synchronization primitives**, such as queues, semaphores, mutexes, and event groups, which enable tasks to exchange data and coordinate their execution. These primitives are created with dynamic memory allocation from the FreeRTOS heap.
- A variety of **APIs** and mechanisms that facilitate memory management operations, such as creating and deleting tasks and queues, allocating and freeing heap memory, and querying available heap space.

Dynamic Memory Allocation and Deallocation

Dynamic memory allocation is the process of requesting and releasing memory at runtime. It allows developers to create data structures and objects whose size and lifetime are not known at compile-time. Dynamic memory allocation is useful in resource-constrained environments because it enables more efficient use of available memory.

FreeRTOS facilitates dynamic memory allocation and deallocation on the ESP32 by providing:

- A **custom memory allocator** that replaces the standard C `malloc()` and `free()` functions. The custom allocator uses a memory region called the FreeRTOS heap to allocate memory for tasks, queues, timers, semaphores, mutexes, event groups, software timers, etc. The size and location of the FreeRTOS heap are defined by the developer at compile-time.
- A set of **APIs** that allow developers to allocate and free memory from the FreeRTOS heap. These APIs include:
  - `pvPortMalloc()`
  - `vPortFree()`
  - `xPortGetFreeHeapSize()`
  - `xPortGetMinimumEverFreeHeapSize()`

These APIs are safe to use from tasks and Interrupt Service Routines (ISRs).

- **Five heap implementations** with varying complexity and features:
  - **Heap_1**: The simplest implementation, which does not support memory deallocation. It can only allocate memory until the heap is exhausted.
  - **Heap_2**: Supports memory allocation and deallocation but does not attempt to coalesce adjacent free blocks into larger blocks. This can lead to memory fragmentation over time.
  - **Heap_3**: Simply wraps the standard C `malloc()` and `free()` functions, relying on the system heap defined by the compiler, which may not be suitable for real-time

applications.
    - **Heap_4**: The recommended implementation, which supports memory allocation and deallocation, and coalesces adjacent free blocks into larger blocks. It uses a linked list to manage heap blocks and critical sections to protect the list from concurrent access.
    - **Heap_5**: Similar to Heap_4 but allows adding multiple non-contiguous memory regions to the heap. This is useful for platforms with memory scattered across different locations.

Developers can choose the desired heap implementation by including the appropriate header file in their project, such as `heap_1.c`, `heap_2.c`, etc.

Potential Issues and Considerations for Dynamic Memory Allocation

Dynamic memory allocation in FreeRTOS has several potential issues and considerations for developers:

- **Memory Leaks**: Occur when dynamically allocated memory blocks are not freed after they are no longer needed. This can reduce the available heap space and eventually lead to memory allocation failures. To avoid memory leaks, developers should always free allocated memory blocks and use heap monitoring tools to detect and debug memory leaks.
- **Memory Corruption**: Occurs when a memory block is accessed or modified after it has been freed, or when a memory block is overwritten by another block. This can cause unexpected behavior and system crashes. To avoid memory corruption, developers should follow good programming practices, such as using pointers carefully, checking the return values of memory allocation functions, and avoiding buffer overflows.
- **Memory Alignment**: Refers to the requirement that memory access must be aligned with the CPU architecture's word size. As mentioned earlier, the ESP32 uses a 32-bit architecture, meaning each word is 4 bytes. If memory access is not aligned with word boundaries, it can increase CPU cycles and cause bus errors. To ensure memory alignment, FreeRTOS aligns all heap blocks to 8-byte boundaries by default. Developers can also use the `portBYTE_ALIGNMENT` macro to specify a different alignment value.

Understanding Memory Fragmentation

Memory fragmentation is a phenomenon where available memory is divided into small, non-contiguous blocks that cannot be used for allocation requests. Memory fragmentation can degrade system performance and stability and even lead to memory allocation failures. Memory fragmentation can be classified into two types: internal and external.

- **Internal Fragmentation**: This occurs when the memory block is larger than the requested size, and the excess space is wasted. For example, if a 16-byte block is allocated for a 10-byte request, 6 bytes are wasted internally. Internal fragmentation can be reduced by using smaller block sizes or combining multiple objects into a single block.
- **External Fragmentation**: This happens when gaps between allocated blocks are too small to satisfy any allocation requests. For instance, if there are three free blocks of 4 bytes, 8 bytes, and 12 bytes, none of them can fulfill a 16-byte request. External fragmentation can be minimized by merging adjacent free blocks into a larger one or using compaction techniques to move allocated blocks and eliminate gaps.

Memory fragmentation can occur in both static and dynamic memory allocation, but it is more common and problematic in dynamic memory allocation. This is because dynamic memory allocation involves frequent requests and releases of variable-sized blocks, creating an irregular pattern of free and used spaces in the heap.

In the context of ESP32 and FreeRTOS, memory fragmentation can happen due to several factors, such as:

- **Use of Different Memory Regions and Types**: ESP32 uses different memory regions (internal and external) and memory types (static and dynamic) to manage its limited RAM and flash resources. These regions and types have different characteristics and constraints that affect their fragmentation levels. For example, internal DRAM has faster access speed but a smaller size than external PSRAM; static memory has fixed size and location but doesn’t experience fragmentation, while dynamic memory has variable size and location but is prone to fragmentation.
- **Heap Implementation Choices**: FreeRTOS provides five different heap implementations that vary in complexity and features. These implementations have different impacts on the internal and external fragmentation levels in the FreeRTOS heap. For example, Heap_1 has no external fragmentation but suffers from high internal fragmentation; Heap_2 has low internal fragmentation but high external fragmentation; Heap_3 has variable fragmentation depending on the system heap; Heap_4 and Heap_5 have low internal and external fragmentation by combining free blocks.
- **Memory Allocation and Deallocation Patterns**: The way tasks allocate and free memory from the FreeRTOS heap can affect fragmentation levels. For example, if tasks allocate and release memory randomly and unpredictably, it can create more gaps and irregularities in the heap. Conversely, if tasks allocate and release memory consistently and regularly, it can create more continuous and uniform blocks in the heap.

Introduction to Queues in FreeRTOS

Queues are one of the inter-task coordination primitives provided by FreeRTOS. A queue is a data structure that holds several items of the same type in a first-in, first-out (FIFO) order. Queues are useful for multitasking applications as they allow tasks to exchange data and synchronize their execution.

The basic concepts of a queue are:

- **Structure**: A queue consists of two parts: the control block and the storage area. The control block contains information about the queue, such as its name, size, item size, the number of items, pointers to the head and tail of the queue, etc. The storage area is a byte array that holds the actual items in the queue. The size of the storage area is determined by multiplying the item size by the queue length.
- **Purpose**: Queues have two main purposes: data transfer and task synchronization. Data transfer refers to the process of sending and receiving data between tasks using a queue. Task synchronization refers to the process of blocking and unblocking tasks based on the availability of data in the queue.
- **Benefits**: Queues offer several benefits for multitasking applications, such as:
  - **Decoupling**: Queues decouple the sending and receiving tasks, meaning they do not need to know each other's identity, priority, or status. They only need to know the queue name they use to communicate.
  - **Buffering**: Queues store data between the sending and receiving tasks, meaning they do not need to be synchronized in time. The sender can send data at any time, and the receiver can receive data at any time, as long as there is space or data in the queue.
  - **Scalability**: Queues can easily scale to support multiple sending and receiving tasks, meaning they can handle simultaneous and varying data flows. Several tasks can share the same queue to send or receive data.

Queue Implementation for Task Communication

To implement queues in FreeRTOS for the ESP32, follow these steps:

1. **Creating a Queue**:
   A queue can be created using the `xQueueCreate()` API function. This function takes two parameters: the queue length (the number of items it can hold) and the size of each item (the number of bytes per item). It returns a handle for the created queue, or `NULL` if creation fails. For example:

   ```
   // Create a queue that can hold 10 items, each 4 bytes in size
   QueueHandle_t xQueue = xQueueCreate(10, sizeof(uint32_t));
   ```

2. **Sending Data to the Queue**:
   Data can be sent to a queue using the `xQueueSend()` or `xQueueSendFromISR()` API functions. These functions take three parameters: the queue handle, a pointer to the data to be sent, and a timeout value (the number of ticks to wait if the queue is full). These functions return `pdTRUE` if data is successfully sent or `pdFALSE` if the timeout expires or an error

occurs. Use `xQueueSend()` from a task, and `xQueueSendFromISR()` from an ISR. For example:

```
// Send the value 100 to the queue from a task
uint32_t ulValueToSend = 100;
BaseType_t xStatus = xQueueSend(xQueue, &ulValueToSend, 0);

// Send the value 200 to the queue from an ISR
uint32_t ulValueToSend = 200;
BaseType_t xStatus = xQueueSendFromISR(xQueue, &ulValueToSend, NULL);
```

3. **Receiving Data from the Queue**:
   Data can be received from a queue using the `xQueueReceive()` or `xQueueReceiveFromISR()` API functions. These functions take three parameters: the queue handle, a pointer to the variable where the received data will be stored, and a timeout value (the number of ticks to wait if the queue is empty). These functions return `pdTRUE` if data is successfully received or `pdFALSE` if the timeout expires or an error occurs. For example:

```
// Receive a value from the queue into a variable from a task
uint32_t ulReceivedValue;
BaseType_t xStatus = xQueueReceive(xQueue, &ulReceivedValue, portMAX_DELAY);

// Receive a value from the queue into a variable from an ISR
uint32_t ulReceivedValue;
BaseType_t xStatus = xQueueReceiveFromISR(xQueue, &ulReceivedValue, NULL);
```

4. **Deleting a Queue**:
   A queue can be deleted using the `vQueueDelete()` API function. This function takes one parameter: the queue handle to be deleted. It frees the memory allocated for the queue and removes it from kernel control. For example:

```
// Delete the queue
vQueueDelete(xQueue);
```

Common Use Cases for Queues in Task Communication

Queues are crucial for effective task communication in various practical use cases, such as:

- **Producer-Consumer**: A common pattern where one or more tasks produce data and send it to a queue, and one or more tasks consume data by receiving it from the queue. For example, a sensor task could read data from a sensor and send it to a queue, while a

display task could receive the data and show it on an LCD screen.
- **Command and Response**: Another common pattern where one task sends commands to another task through a queue, and the other task sends responses back through a separate queue. For instance, a user interface task might send commands to a motor control task via a queue, and the motor control task could send status updates back through another queue.

Queue Synchronization and Data Transfer

Queue synchronization refers to the process of blocking and unblocking tasks based on the availability of data in the queue. This synchronization allows tasks to wait for data to be sent or received without wasting CPU time.

Queue synchronization mechanisms include:

- **Blocking**: When a task tries to send or receive data from a full or empty queue, respectively. The task enters the Blocked state and waits until there is space or data in the queue or until the timeout expires. The task is removed from the Ready list and placed in the Blocked list associated with that queue.
- **Unblocking**: When a blocked task in a queue is able to send or receive data. The task exits the Blocked state and returns to the Ready state. It is removed from the Blocked list and placed in the Ready list based on its priority.
- **Preemption**: When a blocked task in a queue has a higher priority than the currently running task. The currently running task is preempted and placed in the Ready list, and the blocked task is chosen for execution by the scheduler.
- **Yielding**: When a blocked task in a queue has the same priority as the currently running task. The running task voluntarily gives up the CPU to the blocked task by calling the `taskYIELD()` API function. The running task remains in the Ready list but yields the rest of its time slice.

Queue data transfer refers to the process of sending and receiving data between tasks using a queue. This allows tasks to exchange information and coordinate their actions effectively.

Code Sample

This example demonstrates a simple FreeRTOS queue communication between two tasks (`Task1` and `Task2`) running on an ESP32. Here's how it works:

1. `Task1`: This task generates a random integer between 0 and 100, dynamically allocates memory for it using `pvPortMalloc()`, and attempts to send the pointer to the integer to a queue (`xQueue`). If the queue is full and the message cannot be sent, it prints an error message and frees the allocated memory. After each send attempt, the task delays for 1 second.
2. `Task2`: This task waits to receive data from the queue. When a message is received, it prints the received value and then frees the memory used for the integer.
3. `setup()`: Initializes the serial communication, creates the queue, and starts the two tasks (`Task1` and `Task2`) pinned to core 1. If the queue creation fails, it prints an error message.
4. `loop()`: The main Arduino loop remains empty since the tasks are running independently of it.

Here’s the full code:

```
QueueHandle_t xQueue;

void Task1(void *pvParameters) {
  int *p;
  while (1) {
    // Dynamically allocate memory for an integer
    p = (int *)pvPortMalloc(sizeof(int));
    *p = random(0, 100); // Generate a random number between 0 and 100

    // Attempt to send the pointer to the queue, wait indefinitely if needed
    if (xQueueSend(xQueue, &p, portMAX_DELAY) != pdPASS) {
      Serial.println("Failed to post to queue");
      vPortFree(p); // Free memory if message could not be sent to the queue
    }

    vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second
  }
}

void Task2(void *pvParameters) {
```

```
  int *p;
  while (1) {
    // Wait to receive a pointer from the queue, wait indefinitely if needed
    if (xQueueReceive(xQueue, &p, portMAX_DELAY)) {
      Serial.print("Received: ");
      Serial.println(*p); // Print the received value
      vPortFree(p); // Free memory after processing
    }
  }
}

void setup() {
  Serial.begin(115200);

  // Create a queue capable of holding 10 integer pointers
  xQueue = xQueueCreate(10, sizeof(int *));

  if (xQueue == NULL) {
    Serial.println("Error creating the queue");
  }

  // Create two tasks pinned to core 1
  xTaskCreatePinnedToCore(Task1, "Task1", 10000, NULL, 1, NULL, 1);
  xTaskCreatePinnedToCore(Task2, "Task2", 10000, NULL, 1, NULL, 1);

  vTaskDelete(NULL); // Delete the setup task to free memory
}

void loop() {
  // Empty loop since tasks are handled in FreeRTOS tasks
}
```

Key Points:

- **Dynamic Memory Allocation**: Each task dynamically allocates memory for the integer it sends, and the receiving task is responsible for freeing that memory after use.
- **Queue**: A queue is created to hold pointers to integers. Both tasks communicate through this queue.
- **Core Assignment**: The tasks are pinned to core 1 for performance reasons, but this can be changed based on requirements.