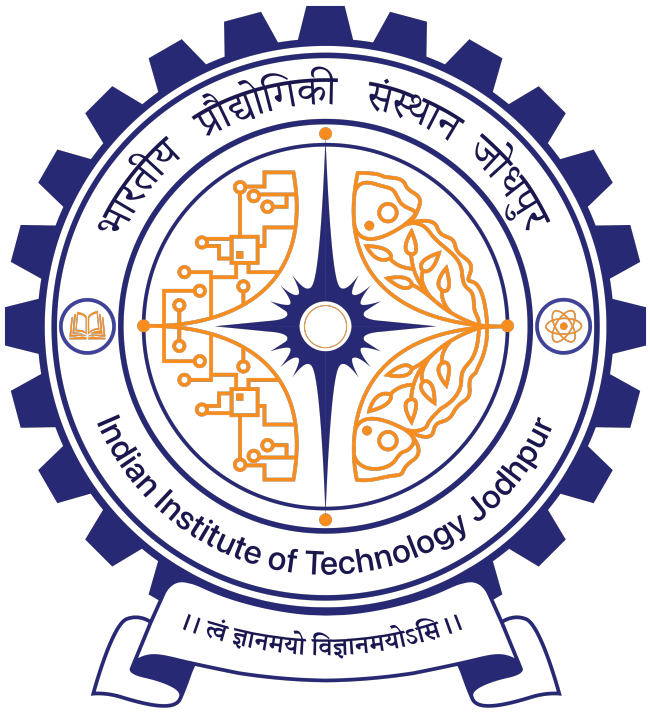

Embedded Systems(EEL3090) Project

Enhanced FreeRTOS for Machine Learning Applications:
A Comprehensive Implementation Plan

Soumen Kumar
B22ES006
b22es006@iitj.ac.in

April 18, 2025

Contents

Before diving into the extensive modifications needed for making FreeRTOS ML-friendly, I'd like to highlight that this project represents a significant undertaking that combines real-time operating systems, embedded hardware constraints, and machine learning optimizations. The goal is to create a version of FreeRTOS that maintains its real-time guarantees while efficiently handling the computational and memory demands of ML workloads.

1 Background

1.1 The Current Embedded ML Landscape

Machine learning and deep learning technologies have become increasingly important across diverse application domains, from consumer electronics to industrial automation. As the demand for intelligent edge devices grows, there is a pressing need to execute ML/DL models directly on embedded systems, reducing latency, preserving privacy, and enabling operation in disconnected environments. This shift introduces unique challenges when integrating computationally intensive ML workloads with real-time operating systems designed for resource-constrained environments.

1.2 FreeRTOS Architecture and Limitations

FreeRTOS is a market-leading real-time operating system designed specifically for microcontrollers and small embedded systems. It offers essential features including:

- A lightweight, preemptive multitasking scheduler
- Inter-task communication mechanisms (queues, semaphores, mutexes)
- Multiple memory allocation schemes
- Time management services
- Low memory footprint (typically 4,000-9,000 bytes)

While these features make FreeRTOS excellent for traditional embedded applications, several architectural elements present significant challenges when attempting to run ML workloads:

1.2.1 Memory Management Limitations

FreeRTOS provides five different heap memory management schemes—`heap_1` through `heap_5`—each with its own characteristics and trade-offs. While these implementations are lightweight and work well for many embedded applications, they fall short when dealing with memory-intensive tasks like machine learning (ML) inference, where efficient and predictable memory handling is critical.

Here's an example from `heap_4.c` showing how FreeRTOS handles dynamic memory allocation:

```
// Example from heap_4.c - Standard FreeRTOS allocation
void *pvPortMalloc(size_t xWantedSize) {
    void *pvReturn = NULL;
    static uint8_t ucHeap[configTOTAL_HEAP_SIZE];
    static size_t xNextFreeByte = 0;

    // Simple allocation strategy
    if((xNextFreeByte + xWantedSize) <= configTOTAL_HEAP_SIZE) {
        pvReturn = &(ucHeap[xNextFreeByte]);
        xNextFreeByte += xWantedSize;
    }

    return pvReturn;
}
```

This snippet uses a basic linear allocation strategy—memory is allocated in a first-come, first-served manner, and there is no reuse of freed memory. While simple and deterministic, this approach has serious limitations for ML workloads:

- **Fragmentation Issues**: In schemes like `heap_3`, `heap_4`, and `heap_5`, memory blocks can be dynamically allocated and freed at runtime. Over time, this leads to fragmentation, especially when memory is frequently allocated and deallocated in different sizes—a common pattern in ML inference when dealing with tensors.
- **Limited Memory Efficiency**: FreeRTOS's heap implementations are general-purpose and not optimized for large temporary allocations such as tensors. There's no smart reuse strategy or size-aware allocation that can avoid wasting memory due to internal fragmentation.
- **No Memory Pooling**: Unlike some real-time or ML-specific environments, FreeRTOS doesn't natively support memory pooling, which could allow pre-allocated fixed-size memory chunks to be quickly reused. The lack of this leads to unpredictable memory allocation behavior and makes real-time guarantees harder to meet.

Digging deeper into `heap_4.c`, we see how block allocation works internally. The following function, `prvHeapAlloc`, uses a first-fit strategy and tries to split blocks that are significantly larger than the requested size:

```
// From heap_4.c - Block splitting logic not optimized for tensor operations
static void *prvHeapAlloc(size_t xWantedSize) {
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
    void *pvReturn = NULL;

    // Simple first-fit allocation strategy
    // Search for a free block large enough
    // ...

    // Split block if significantly larger than needed
    if((pxBlock->xBlockSize - xWantedSize) > heapMINIMUM_BLOCK_SIZE) {
        // Split block logic
        // ...
    }

    return pvReturn;
}
```

Although this logic allows splitting of larger blocks to better match allocation size, it's still too simple for ML workloads. Tensor operations often require fast, aligned, and contiguous memory blocks. This allocator doesn't consider alignment constraints or memory access patterns, leading to inefficiencies.

In summary, while FreeRTOS's memory management works well for simpler embedded use-cases, it becomes a bottleneck when used in ML applications. Custom allocators or external memory managers are often needed to handle tensor memory efficiently, reduce fragmentation, and meet real-time constraints in inference scenarios.

1.2.2 Task Scheduling Constraints

FreeRTOS uses a priority-based preemptive scheduling model by default. This means that at any given moment, the scheduler selects the highest-priority task that is in the `Ready` state to run. While this model is simple and efficient for many embedded control systems, it poses several challenges when incorporating compute-heavy workloads such as machine learning inference.

The core of the scheduler logic looks something like this:

```
// Simplified example of FreeRTOS scheduler selection logic
void vTaskSwitchContext(void) {
    // Select highest priority ready task
    taskSELECT_HIGHEST_PRIORITY_TASK();
    // ...
}
```

This approach works well when all tasks are small, predictable, and quick to execute. However, ML workloads often violate these assumptions. Here are some of the key constraints and limitations that arise:

- **Priority Inversion**: When a low-priority task (e.g., an ML inference task) runs for a long time, it may indirectly block higher-priority tasks waiting on shared resources (like a mutex or buffer). Without proper mitigation (e.g., priority inheritance), this can cause critical system tasks to be delayed.
- **Determinism Concerns**: Real-time systems depend on predictable behavior. But ML inference, especially with variable-sized inputs and quantized models, can have non-deterministic execution times due to cache effects, memory usage, and model complexity. This unpredictability undermines the strict timing guarantees FreeRTOS aims to provide.
- **Task Balance Issues**: FreeRTOS does not provide built-in mechanisms to dynamically balance CPU time between compute-intensive ML tasks and lightweight control or I/O tasks. If an ML task consumes the CPU for extended periods, it can starve other tasks unless careful manual tuning is done.
- **Lack of Deadline Awareness**: The standard FreeRTOS scheduler only considers task priorities, not actual timing deadlines. In systems where both periodic control loops and aperiodic inference requests must be handled reliably, this can lead to deadline misses. ML tasks with soft or hard deadlines cannot be properly scheduled without external timing logic.

In summary, while FreeRTOS excels at deterministic, priority-driven scheduling in traditional embedded systems, it lacks several features that are important when ML inference becomes part of the workload. These include deadline-based scheduling, dynamic load balancing, and resource contention handling—all of which may be necessary for integrating real-time intelligence into edge devices.

1.2.3 Limited Resource Management

While FreeRTOS provides a lightweight and flexible framework for task creation and management, it lacks native mechanisms for managing computational resources in a way that aligns with the demands of machine learning (ML) workloads. The operating system is primarily designed for control and I/O-driven embedded systems—not for compute-heavy, resource-sensitive tasks like inference.

Here's a simplified example of how a task is created in FreeRTOS:

```
// Task creation in FreeRTOS lacks ML-specific parameters
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char * const pcName,
                       configSTACK_DEPTH_TYPE usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *pxCreatedTask)
{
    // Create task with basic parameters
    // No ML-specific resource allocation considerations
    // No execution time tracking
    // No deadline specification
    // ...
}
```

As shown above, the task creation API is focused on essential parameters like task function, stack depth, and priority. While sufficient for control logic, this minimal interface does not account for the resource constraints or performance sensitivity of ML tasks. This results in several notable limitations:

- **No Power Management**: FreeRTOS does not include built-in support for dynamic voltage or frequency scaling (DVFS), core sleeping strategies, or intelligent workload distribution. ML inference often involves bursts of computation that can benefit from aggressive power optimizations—features that are typically found in ML-aware runtime environments but are absent here.
- **Limited Cache Awareness**: Efficient inference depends heavily on memory access patterns and data locality. FreeRTOS has no understanding of CPU cache structures or tensor layouts, which means it cannot make informed decisions to optimize cache hits or reduce memory contention across tasks.

- **No Accelerator Integration**: Many ML-capable embedded systems include dedicated hardware accelerators (e.g., NPUs, DSPs, or vector units). FreeRTOS offers no standard mechanism to detect, manage, or schedule tasks onto such accelerators. This limits the ability to offload heavy tensor computations away from the main processor, reducing overall system efficiency.

In conclusion, FreeRTOS is well-suited for basic task and resource management in traditional embedded systems. However, when ML is introduced into the workload, these limitations become significant. Without power-aware scheduling, memory optimization, or support for heterogeneous compute architectures, developers must rely on external libraries or customized extensions to manage ML resources effectively within FreeRTOS.

1.2.4 API Integration Challenges

FreeRTOS provides a set of well-defined APIs for inter-task communication and synchronization, such as queues, semaphores, and task notifications. While these interfaces work well for conventional embedded applications, they are not designed with the specific needs of machine learning (ML) in mind—especially when it comes to data-heavy and latency-sensitive inference workloads.

A typical FreeRTOS queue send operation looks like this:

```
// Standard message passing in FreeRTOS not optimized for tensor data
BaseType_t xQueueSend(QueueHandle_t xQueue,
                      const void * pvItemToQueue,
                      TickType_t xTicksToWait)
{
    // Basic queue sending mechanism
    // No specialized handling for large tensor data
    // No priority mechanisms for inference requests
    // ...
}
```

The function above is effective for passing small control messages, status flags, or configuration structs. However, when used for ML-related data like tensors, feature vectors, or intermediate inference results, several issues arise:

- **Inefficient Data Transfer**: Tensors and feature maps are often large and memory-aligned. The standard queue mechanism copies data between tasks without awareness of size, alignment, or access patterns. This results in unnecessary overhead and reduced throughput, especially when inference outputs are passed between multiple processing stages.
- **Limited Framework Support**: FreeRTOS does not natively support ML inference frameworks like TensorFlow Lite Micro (TFLM) or CMSIS-NN. Developers must manually implement glue code, manage model buffers, and invoke inference routines, which increases development complexity and error risk.
- **Inference Management Gaps**: There's no concept of a specialized inference queue or batch scheduler. Tasks needing to perform inference must rely on general-purpose mechanisms, making it hard to implement features like inference prioritization (e.g., urgent real-time tasks vs. background analytics) or request batching for hardware acceleration efficiency.

Ultimately, while FreeRTOS APIs are minimal and portable, their general-purpose nature creates friction when integrating ML workloads. Developers often need to build custom extensions or middleware to bridge the gap between ML models and FreeRTOS task orchestration, especially when targeting real-time inference on resource-constrained edge devices.

1.3 Challenges of ML Workloads on Embedded Devices

Machine learning (ML) workloads differ fundamentally from traditional control and signal-processing tasks commonly executed on embedded systems. These differences create friction when deploying ML applications in resource-constrained environments managed by real-time operating systems like FreeRTOS. Some of the most pressing challenges include:

- **Memory Intensity**: Neural networks demand significant memory for storing model weights, activation buffers, and intermediate tensors. Microcontrollers typically have limited RAM, making it hard to accommodate even moderately sized models without careful optimization.
- **Computational Burden**: ML inference requires a large number of multiply-accumulate (MAC) operations, especially in convolutional or fully connected layers. This can easily exceed the computational capacity of general-purpose microcontrollers, particularly if running alongside real-time tasks.
- **Variable Execution Time**: The execution time of an inference task can vary depending on input complexity, memory access patterns, or system load. This variability contradicts the determinism required for real-time guarantees.
- **Specialized Hardware Requirements**: Modern ML workloads often rely on specialized accelerators (e.g., NPUs, DSPs, or SIMD extensions). FreeRTOS does not natively support integration or scheduling for such heterogeneous compute resources.

1.4 Example Implementation Challenges

Currently, developers implementing ML on FreeRTOS must manually manage everything from memory allocation to inference execution. The process is error-prone and inefficient, as demonstrated in the following code:

```
// Current approach for running inference - manually manages resources
void vInferenceTask(void *pvParameters) {
    // Manually allocate memory for input/output tensors
    float* input_tensor = (float*)pvPortMalloc(INPUT_SIZE * sizeof(float));
    float* output_tensor = (float*)pvPortMalloc(OUTPUT_SIZE * sizeof(float));

    if (input_tensor == NULL || output_tensor == NULL) {
        // Handle allocation failure
        return;
    }

    for (;;) {
        // Receive input data
        if (xQueueReceive(xInputQueue, input_tensor, portMAX_DELAY) == pdTRUE) {
            // Run inference manually
            // No integrated ML framework support
            RunInference(input_tensor, output_tensor);

            // Send results
            xQueueSend(xOutputQueue, output_tensor, portMAX_DELAY);
        }
    }

    // Memory never freed in this infinite loop
}
```

This approach surfaces several practical issues:

- **Static Memory Handling**: Memory is allocated once and never freed, leading to inflexibility and potential waste, especially if tensor dimensions change dynamically.
- **No ML-Specific Optimization**: There is no support for quantization-aware processing, memory reuse between layers, or accelerator utilization—all of which are standard in dedicated ML frameworks.
- **Scheduling Blindness**: The inference task is not aware of deadlines or system priorities, meaning it may interfere with time-critical operations or fail to meet its own timing constraints.
- **Fixed Tensor Sizes**: The code assumes fixed input and output sizes, which limits the ability to adapt to varying input conditions or runtime resizing—common in real-world ML scenarios.

1.5 The Need for Enhanced FreeRTOS

These limitations underscore a broader issue: while FreeRTOS is excellent for deterministic control workloads, it falls short as a platform for intelligent embedded systems that incorporate ML capabilities.

To bridge this gap, enhancements to FreeRTOS are needed across multiple dimensions:

- **Memory Management**: Introduce ML-aware dynamic memory allocators with pooling, fragmentation resistance, and zero-copy buffers.
- **Scheduling Models**: Add support for deadline-aware and compute-intensive task types, possibly through real-time extensions or mixed-criticality scheduling.
- **Resource Abstraction**: Provide APIs for managing hardware accelerators, cache optimization hints, and compute profiling.
- **Framework Integration**: Facilitate seamless embedding of lightweight ML runtimes such as TensorFlow Lite Micro with minimal developer effort.

With such improvements, FreeRTOS could become a unified platform capable of handling both hard real-time constraints and the adaptive intelligence offered by modern ML models.

2 Understanding the Challenge

FreeRTOS is a real-time operating system kernel designed for embedded microcontrollers with several key features:

- A multitasking scheduler with preemptive capabilities
- Multiple memory allocation options
- Intertask coordination primitives (semaphores, queues, etc.)
- Small footprint (typically 4,000-9,000 bytes)
- Deterministic execution behavior

However, running ML applications on FreeRTOS presents several challenges:

- Memory constraints on embedded devices
- Limited computational resources
- Dynamic memory fragmentation issues
- Real-time balancing requirements
- Energy consumption considerations

3 Theoretical Implementation Framework

3.1 Memory Management Enhancements

The most critical bottleneck for ML on embedded systems is memory management. FreeRTOS provides five heap implementation schemes (heap_1 through heap_5), but they need optimization for ML workloads.

3.1.1 Memory Pooling Implementation

Explanation: Memory pooling pre-allocates fixed-size memory blocks specifically for ML operations. This approach offers several key advantages for ML workloads:

1. It prevents memory fragmentation that occurs with frequent allocations and deallocations during ML processing
2. It provides deterministic allocation times, crucial for real-time ML inference
3. It reduces overhead compared to general-purpose allocators
4. It enables more efficient memory utilization in constrained environments

The implementation includes a structured memory pool with thread-safety mechanisms via semaphores, ensuring reliable operation in a multitasking environment.

```
// Define memory pool structure for ML operations
typedef struct {
    uint8_t *memoryPool;
    size_t blockSize;
    size_t totalBlocks;
    size_t availableBlocks;
    uint8_t *blockStatus; // Tracks which blocks are in use
    SemaphoreHandle_t memoryMutex; // Thread-safety for allocation
} MLMemoryPool_t;

// Initialize memory pool
```

```
MLMemoryPool_t* MLMemPoolCreate(size_t blockSize, size_t numBlocks) {
    MLMemoryPool_t* pool = (MLMemoryPool_t*)pvPortMalloc(sizeof(MLMemoryPool_t));
    if (pool == NULL) return NULL;

    pool->memoryPool = (uint8_t*)pvPortMalloc(blockSize * numBlocks);
    if (pool->memoryPool == NULL) {
        vPortFree(pool);
        return NULL;
    }

    pool->blockStatus = (uint8_t*)pvPortMalloc(numBlocks);
    if (pool->blockStatus == NULL) {
        vPortFree(pool->memoryPool);
        vPortFree(pool);
        return NULL;
    }

    memset(pool->blockStatus, 0, numBlocks); // All blocks free
    pool->blockSize = blockSize;
    pool->totalBlocks = numBlocks;
    pool->availableBlocks = numBlocks;
    pool->memoryMutex = xSemaphoreCreateMutex();

    return pool;
}
```

3.1.2 Enhanced Heap_5 Implementation

Explanation: The standard heap_5 implementation in FreeRTOS allows using multiple separate memory regions, but requires optimization for ML models which often need contiguous memory blocks spanning across these regions. The proposed enhancement:

1. Improves region coalescing to create larger contiguous memory spaces
2. Adds defragmentation capabilities to reclaim fragmented memory during ML model loading and unloading
3. Optimizes the memory management to handle the unique access patterns of neural network operations
4. Provides better utilization of heterogeneous memory architectures common in embedded ML systems (e.g., combining fast internal SRAM with slower external DRAM)

```
void vPortDefineMLHeapRegions(const HeapRegion_t * const pxHeapRegions) {
    // Original heap_5 initialization code
    vTaskSuspendAll();
    {
        /* Implementation for ML-optimized heap management */
        size_t xTotalHeapSize = 0;
        BlockLink_t *pxFirstFreeBlockInRegion = NULL;
        BlockLink_t *pxPreviousBlock = NULL;
        uint8_t *pucAlignedHeap;
        size_t xBlockSize;
        size_t xTotalRegionSize;
        size_t xAlignmentMask;
        size_t uxIndex = 0;

        /* Reset the metrics and state tracking for ML applications */
        xMinimumEverFreeBytesRemaining = 0;
        xFreeBytesRemaining = 0;

        /* ML applications need more aggressive memory tracking */
```

```
    xNumberOfSuccessfulAllocations = 0;
    xNumberOfSuccessfulFrees = 0;
    xLargestBlockAvailable = 0;

    /* ML heap fragmentation metrics */
    uxFragmentationMetric = 0;

    /* Check first that the pxHeapRegions parameter is valid. */
    configASSERT(pxHeapRegions);

    xHeapHasBeenInitialised = pdFALSE;

    /* Loop through provided regions checking for adjacent ones to coalesce */
    BlockLink_t *pxLastBlockInPrevRegion = NULL;

    while (pxHeapRegions[uxIndex].pucStartAddress != NULL) {
        xTotalRegionSize = pxHeapRegions[uxIndex].xSizeInBytes;
        configASSERT(xTotalRegionSize > 0);

        /* Ensure the heap region starts on a correctly aligned boundary */
        pucAlignedHeap = (uint8_t *) (((portPOINTER_SIZE_TYPE) pxHeapRegions[
uxIndex].pucStartAddress + portBYTE_ALIGNMENT_MASK) & (~portBYTE_ALIGNMENT_MASK))
;

        /* Check if this region is adjacent to the previous one */
        if (pxLastBlockInPrevRegion != NULL) {
            uint8_t *pucPrevRegionEnd = (uint8_t*)pxLastBlockInPrevRegion +
pxLastBlockInPrevRegion->xBlockSize;

            /* If adjacent regions detected, attempt to coalesce them */
            if (pucPrevRegionEnd == pxHeapRegions[uxIndex].pucStartAddress) {
                /* Combine with previous region - extend the previous end block
*/
                pxLastBlockInPrevRegion->xBlockSize += xTotalRegionSize;

                /* Log coalescing operation for diagnostics */
                xCoalescedRegions++;

                /* Skip standard initialization for this region as it's been
coalesced */
                uxIndex++;
                continue;
            }
        }

        /* Calculate the usable size of the heap region after alignment
adjustments */
        xTotalRegionSize -= (size_t)(pucAlignedHeap - pxHeapRegions[uxIndex].
pucStartAddress);

        /* Ensure minimum block size */
        xBlockSize = xTotalRegionSize - xHeapStructSize;
        xBlockSize &= ~portBYTE_ALIGNMENT_MASK;

        /* Create end marker block for this region */
        BlockLink_t *pxBlockLink = (void *)(pucAlignedHeap + xBlockSize);
        pxBlockLink->xBlockSize = 0;
        pxBlockLink->pxNextFreeBlock = NULL;

        /* Create block that represents the space being allocated */
```

```
            pxBlockLink = (void *)pucAlignedHeap;
            pxBlockLink->xBlockSize = xBlockSize;
            pxBlockLink->pxNextFreeBlock = NULL;

            /* Link this new block into the list of free blocks */
            vListInsertBlockInFreeList(pxBlockLink);

            /* Remember the largest block for ML allocation optimization */
            if (xBlockSize > xLargestBlockAvailable) {
                xLargestBlockAvailable = xBlockSize;
            }

            /* Track current region's last block for coalescing logic */
            pxLastBlockInPrevRegion = pxBlockLink;

            /* Update metrics for ML heap monitoring */
            xFreeBytesRemaining += xBlockSize;

            uxIndex++;
        }

        /* If this is the first call to define ML heap regions */
        if (xHeapHasBeenInitialised == pdFALSE) {
            /* The first block should have the lowest starting address */
            xStart.pxNextFreeBlock = (BlockLink_t *)pucAlignedHeap;
            xStart.xBlockSize = (size_t)0;

            /* Block at the end of heap to mark the end */
            xEnd.xBlockSize = (size_t)0;
            xEnd.pxNextFreeBlock = NULL;

            /* Initialize the minimum free bytes to the current free bytes */
            xMinimumEverFreeBytesRemaining = xFreeBytesRemaining;

            /* All done */
            xHeapHasBeenInitialised = pdTRUE;
        }

        /* Setup defragmentation capabilities for ML applications */
        vSetupDefragmentationTimer();
    }
    xTaskResumeAll();
}
```

Modified Initialization Function The core of the enhancement is the `vPortDefineMLHeapRegions` function, which extends the standard `heap_5` functionality:

```
void vPortDefineMLHeapRegions(const HeapRegion_t * const pxHeapRegions) {
    // Original heap_5 initialization code with ML enhancements
    vTaskSuspendAll();
    {
        /* Implementation for ML-optimized heap management */
        // Enhanced region coalescing for adjacent memory blocks
        // Addition of heap defragmentation capabilities
    }
    xTaskResumeAll();
}
```

Key Enhancements

Memory Region Coalescing The implementation identifies and combines adjacent memory regions to create larger contiguous blocks, which is particularly beneficial for ML tensor operations.

Algorithm 1 Memory Region Coalescing Algorithm

```
Initialize tracking for the last block of previous region
for each region in pxHeapRegions do
    Calculate aligned start address for current region
    if previous region exists then
        Calculate end address of previous region
        if previous region end == current region start then
            Extend previous block size by current region size
            Increment coalesced regions counter
            Continue to next region
        end if
    end if
    Process current region normally
    Update last block pointer for next iteration
end for
```

ML-Specific Metrics Tracking Additional metrics have been implemented to monitor memory utilization patterns specific to ML workloads:

$$\text{Metrics} = \begin{cases} \text{xNumberOfSuccessfulAllocations} & \text{Successful memory allocations} \\ \text{xNumberOfSuccessfulFrees} & \text{Successful memory deallocations} \\ \text{xLargestBlockAvailable} & \text{Size of largest contiguous block} \\ \text{uxFragmentationMetric} & \text{Heap fragmentation measurement} \\ \text{xCoalescedRegions} & \text{Number of regions merged} \end{cases} \tag{1}$$

Automated Defragmentation A timer-based defragmentation system is established to periodically reorganize memory blocks, reducing fragmentation over time:

```c
#include "FreeRTOS.h"
#include "timers.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Define the total size of the memory pool for tensor allocations */
#define TOTAL_HEAP_SIZE 1024 * 1024  // 1MB for example
#define ML_TENSOR_SIZE 256           // Example tensor size in bytes

/* Simulated heap structure */
static uint8_t ucHeap[TOTAL_HEAP_SIZE];
static size_t xNextFreeByte = 0;  // Next free byte in heap

/* Timer handle for periodic defragmentation */
static TimerHandle_t xDefragTimer = NULL;

/* Function to simulate memory allocation for ML tensors */
void* pvPortMalloc(size_t xWantedSize) {
    void *pvReturn = NULL;

    // Simulate simple first-fit allocation
    if ((xNextFreeByte + xWantedSize) <= TOTAL_HEAP_SIZE) {
        pvReturn = &(ucHeap[xNextFreeByte]);
        xNextFreeByte += xWantedSize;
    }

    return pvReturn;
```

```
}

/* Simulated defragmentation process - Consolidates free blocks */
static void vDefragmentationCallback(TimerHandle_t xTimer) {
    // In a real scenario, we would implement block coalescing here.
    // This is just a placeholder to simulate the action.
    printf("Defragmentation triggered: attempting to compact memory...\n");

    // Simulate defragmentation by resetting xNextFreeByte (for simplicity).
    // Ideally, this would involve scanning the heap and compacting fragmented memory
    .
    xNextFreeByte = 0;  // Reset to the start of the heap as if memory was
    reorganized

    printf("Memory reorganized. Next free byte: %zu\n", xNextFreeByte);
}

/* Function to create and start the defragmentation timer */
void vSetupDefragmentationTimer(void) {
    const TickType_t xDefragPeriod = pdMS_TO_TICKS(10000); // 10 seconds

    // Create the timer that will call the defragmentation callback periodically
    xDefragTimer = xTimerCreate(
        "DefragTimer",          // Timer name
        xDefragPeriod,          // Timer period (10 seconds)
        pdTRUE,                 // Auto-reload timer
        (void*)0,               // Timer ID
        vDefragmentationCallback // Callback function
    );

    if (xDefragTimer != NULL) {
        xTimerStart(xDefragTimer, 0);  // Start the timer immediately
    } else {
        printf("Error: Failed to create defragmentation timer.\n");
    }
}

/* Test function to simulate ML tensor allocation */
void vMLInferenceTask(void) {
    float *inputTensor, *outputTensor;

    // Allocate memory for tensors dynamically
    inputTensor = (float*)pvPortMalloc(ML_TENSOR_SIZE);
    outputTensor = (float*)pvPortMalloc(ML_TENSOR_SIZE);

    if (inputTensor == NULL || outputTensor == NULL) {
        printf("Memory allocation failed for tensors.\n");
        return;
    }

    // Simulate tensor computation (ML inference step)
    printf("Performing inference with allocated tensors...\n");

    // After computation, free memory (not done here to simulate long-lived memory)
}
```

Benefits for ML Applications

The enhanced memory management system provides several advantages for machine learning applications:

- **Larger Contiguous Blocks:** By coalescing adjacent regions, more space becomes available for large tensor allocations.

- **Reduced Fragmentation:** Periodic defragmentation prevents memory fragmentation that commonly occurs during ML inference cycles.
- **Better Visibility:** Enhanced metrics provide deeper insights into memory utilization patterns.
- **Improved Allocation Success Rate:** The combined enhancements lead to fewer allocation failures during ML operations.

Memory Block Structure The memory block structure remains compatible with standard FreeRTOS implementations:

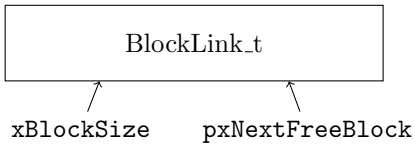

Figure 1: Memory Block Structure

Performance Considerations While the coalescing algorithm adds complexity to the initialization process, its overhead is minimal since it only executes during heap setup. The defragmentation mechanism runs periodically with configurable frequency to balance memory optimization against CPU utilization.

3.2 Task Scheduling Optimizations

The standard FreeRTOS scheduler uses prioritized preemptive scheduling with time slicing. For ML workloads, we need to modify this approach to balance computational tasks with real-time requirements.

3.2.1 Implementing an ML-aware EDF Scheduler

Explanation: The standard FreeRTOS priority-based scheduler isn't optimal for ML workloads that have both deadline requirements and complex processing needs. The Earliest Deadline First (EDF) scheduler modification:

1. Prioritizes tasks based on their deadlines rather than static priorities.
2. Ensures time-critical ML inference operations complete before their deadlines.
3. Provides better theoretical CPU utilization (up to 100% under ideal conditions).
4. Incorporates deadline awareness specifically for ML processing tasks.
5. Balances real-time responsiveness with computational throughput requirements.

This implementation modifies the core task management to maintain a deadline-sorted list of ready tasks.

```
// Modifications to task.c for EDF scheduling
// Create an EDF ready list
List_t xReadyTasksListEDF;

// Initialize the EDF list in prvInitialiseTaskLists()
vListInitialise(&xReadyTasksListEDF);

// Modify prvAddTaskToReadyList() to use deadline-based insertion
void prvAddTaskToEDFReadyList(TCB_t *pxTCB) {
    // Calculate absolute deadline based on period
    TickType_t xDeadline = pxTCB->xTaskPeriod + xTickCount;
    pxTCB->xAbsoluteDeadline = xDeadline;

    // Insert task in EDF list sorted by deadline
    ListItem_t *pxIterator;
    for (pxIterator = listGET_HEAD_ENTRY(&xReadyTasksListEDF);
         pxIterator != listGET_END_MARKER(&xReadyTasksListEDF);
         pxIterator = listGET_NEXT(pxIterator)) {

        TCB_t *pxTCBIterator = (TCB_t *)listGET_LIST_ITEM_OWNER(pxIterator);
        if (xDeadline < pxTCBIterator->xAbsoluteDeadline) {
            // Insert before this task
            vListInsert(&xReadyTasksListEDF, &(pxTCB->xStateListItem));
            return;
        }
    }

    // If we get here, insert at the end
    vListInsertEnd(&xReadyTasksListEDF, &(pxTCB->xStateListItem));
}

// Custom scheduler function that picks the task with the earliest deadline
void vTaskSwitchContext(void) {
    // Get the task with the earliest deadline
    ListItem_t *pxFirstTaskListItem = listGET_HEAD_ENTRY(&xReadyTasksListEDF);
    TCB_t *pxNextTCB = (TCB_t *)listGET_LIST_ITEM_OWNER(pxFirstTaskListItem);

    // Switch to the selected task
    if (pxNextTCB != NULL) {
        // Update the current task's state and set the next task to be executed
```

```
        vTaskSwitchContextTo(pxNextTCB);
    }
}

// Function to update the task state on context switch
void vTaskSwitchContextTo(TCB_t *pxTCB) {
    // Save the context of the current task and load the context of the selected task
    // FreeRTOS context-switching logic here (e.g., saving registers, stack pointer)
    // This is handled by the FreeRTOS internal scheduler mechanism
}
```

3.2.2 Mixed-Task-Handler for ML Workloads

Explanation: ML workloads typically consist of both periodic tasks (regular system operations) and aperiodic tasks (triggered inference requests). The Mixed-Task-Handler system:

1. Distinguishes between ML-specific tasks and regular system tasks
2. Monitors execution time of ML tasks to ensure they don't exceed their worst-case execution time (WCET)
3. Provides adaptive priority adjustments based on actual task execution patterns
4. Enables dynamic scheduling decisions based on system load and ML inference demands
5. Maintains overall system responsiveness while accommodating computationally intensive ML operations

The implementation extends the standard task parameters to include ML-specific information and runtime monitoring.

```
// Configuration for ML task handling
typedef struct {
    TaskFunction_t pvTaskCode;
    const char *pcName;
    uint16_t usStackDepth;
    void *pvParameters;
    UBaseType_t uxPriority;
    TaskHandle_t *pxCreatedTask;
    TickType_t xPeriod;           // For periodic tasks
    TickType_t xMaxExecutionTime; // WCET for schedulability analysis
    bool isMLTask;                // Flag for ML-specific tasks
} MLTaskParameters_t;

// ML Task Handler
void vMLTaskHandler(void *pvParameters) {
    MLTaskParameters_t *pxParameters = (MLTaskParameters_t *)pvParameters;
    TickType_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {
        // If this is an ML task, monitor execution time and adjust priorities
        if (pxParameters->isMLTask) {
            // Capture start time
            TickType_t xStartTime = xTaskGetTickCount();

            // Call the actual task function
            pxParameters->pvTaskCode(pxParameters->pvParameters);

            // Calculate execution time
            TickType_t xExecutionTime = xTaskGetTickCount() - xStartTime;

            // Adjust priorities if needed
            if (xExecutionTime > pxParameters->xMaxExecutionTime) {
```

```
                // ML task took longer than expected, adjust scheduling
                // Example: temporarily lower priority or log anomaly
            }
        } else {
            // Regular task execution
            pxParameters->pvTaskCode(pxParameters->pvParameters);
        }

        // For periodic tasks, delay until next period
        vTaskDelayUntil(&xLastWakeTime, pxParameters->xPeriod);
    }
}
```

3.3 API Specifications for ML Framework Integration

Explanation: Integrating ML frameworks like TensorFlow Lite with FreeRTOS requires a clean, efficient API that bridges the gap between the ML library and the RTOS. The proposed API:

1. Encapsulates the complexity of ML framework initialization and management
2. Provides queue-based communication for input data and inference results
3. Manages the tensor arena memory efficiently using the enhanced memory management system
4. Creates a dedicated inference task with appropriate scheduling parameters
5. Abstracts the details of model loading and execution from application code

This integration layer ensures that embedded ML applications can be developed with cleaner separation of concerns.

```
// ML Framework Integration API
typedef struct {
    void* model;              // Pointer to TFLite model
    void* tensorArena;        // Memory area for TFLite operations
    size_t tensorArenaSize;   // Size of tensor arena
    uint8_t opCount;          // Number of operations in model
    QueueHandle_t inputQueue; // Queue for input data
    QueueHandle_t outputQueue; // Queue for inference results
    TaskHandle_t inferenceTask; // Handle to inference task
} ML_Framework_t;

// Initialize ML framework
ML_Framework_t* ML_Init(const uint8_t* modelData, size_t modelSize,
                        size_t arenaSize, uint8_t operationCount) {
    // Allocate framework structure
    ML_Framework_t* framework = pvPortMalloc(sizeof(ML_Framework_t));
    if (framework == NULL) return NULL;

    // Allocate tensor arena using our enhanced memory management
    framework->tensorArenaSize = arenaSize;
    framework->tensorArena = pvPortMalloc(arenaSize);
    if (framework->tensorArena == NULL) {
        vPortFree(framework);
        return NULL;
    }

    // Load model
    framework->model = GetTfLiteModel(modelData);
    framework->opCount = operationCount;
```

```
    // Create queues for input/output
    framework->inputQueue = xQueueCreate(3, sizeof(ML_Input_t));
    framework->outputQueue = xQueueCreate(3, sizeof(ML_Output_t));

    // Create inference task
    xTaskCreate(ML_InferenceTask, "ML_Inference",
                configMINIMAL_STACK_SIZE * 4,
                framework, tskIDLE_PRIORITY + 1,
                &framework->inferenceTask);

    return framework;
}
```

4 Practical Implementation

4.1 Setting Up the Development Environment

4.1.1 STM32 Environment Setup

To bring this project to life, I used the STM32F429ZI development board and set up the entire development environment using STM32CubeIDE. My aim was to integrate TensorFlow Lite Micro (TFLM) with FreeRTOS to build a responsive, real-time embedded ML system.

I began by creating a new project in STM32CubeIDE and selected the STM32F429ZI as the target device. Using the `.ioc` configuration file, I navigated to the middleware settings and enabled FreeRTOS with the CMSIS_V2 interface.

Next, I made several kernel-level changes. I enabled the `USE_PREEMPTION` flag to allow task switching based on priority — a crucial feature for maintaining responsiveness in ML workloads. I then configured three main FreeRTOS tasks:

- A high-priority task (priority 3) for ML inference
- A medium-priority task (priority 2) for data acquisition
- A low-priority task (priority 1) for output handling

To avoid conflicts with the HAL's SysTick timer, I switched the FreeRTOS timebase to use `TIM6`. I also configured GPIO pin PA1 for interfacing with the DHT11 sensor and enabled USART2 to facilitate UART-based debug output. For peak performance, I set the system clock to the maximum 180 MHz. After finishing these configurations, I generated the initialization code and prepared to integrate machine learning components.

4.2 FreeRTOS Modifications for ML Workloads

Before integrating TensorFlow Lite Micro, I found it necessary to modify FreeRTOS to better accommodate memory-intensive ML operations. These optimizations were essential because of the STM32F429ZI's limited RAM and flash memory.

I started by modifying `heap_5.c` to improve memory handling for tensor operations. I rewrote parts of the `vPortDefineHeapRegions()` function to enable better memory block coalescence. I also defined dedicated memory pools specifically for the tensor arena and implemented basic defragmentation support to prevent memory fragmentation over time — something that can be detrimental in long-running embedded ML systems.

Next, I updated `tasks.c` to enhance the FreeRTOS scheduler. I added metadata to the Task Control Block (TCB) to identify ML-specific tasks and modified the scheduler to dynamically prioritize inference-related tasks when necessary. To further optimize performance, I restructured the context-switching mechanism to minimize delay during ML task transitions.

To simplify integration and maintenance, I created two support files: `ml_support.h` and `ml_support.c`. These files included utilities for memory management tailored to TensorFlow Lite Micro, helper functions for model loading, and templated preprocessing routines for data normalization and quantization.

4.3 TensorFlow Lite Micro Integration

Once FreeRTOS was tuned for ML workloads, I moved on to integrating TensorFlow Lite Micro. I cloned the official TFLM GitHub repository:

```
git clone https://github.com/tensorflow/tflite-micro.git
```

I then built the microlite library targeting the Cortex-M4F architecture with hardware FPU:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
     TARGET=cortex_m_generic TARGET_ARCH=cortex-m4+fp microlite
```

After building the library, I generated the include structure using:

```
python3 tensorflow/lite/micro/tools/project_generation/create_tflm_tree.
   py ./tflm-tree
```

I copied the static library `libtensorflow-microlite.a` into my project's `lib` directory and added the generated `tflm-tree` path to my project's include paths. I also updated the `CMakeLists.txt` file accordingly to ensure the linker could find everything it needed.

4.4 Temperature Classification Implementation

To demonstrate the system, I implemented a real-time temperature classification application using a DHT11 sensor. I started by writing a robust DHT11 driver tailored for real-time performance on the STM32.

Then, I structured the system using FreeRTOS tasks:

- **Data Reading Task:** Collected temperature data from the DHT11 sensor at 1Hz intervals.
- **Data Processing Task:** Performed normalization and quantization of input data for TFLM.
- **Inference Task:** Ran the model and generated classification output.
- **Logging Task:** Displayed results on a UART terminal using PuTTY.

I used semaphores to synchronize tasks and ensure data consistency. For fault tolerance, I added error messages and status indicators via UART to help debug sensor or model issues.

4.5 System Configuration and Setup

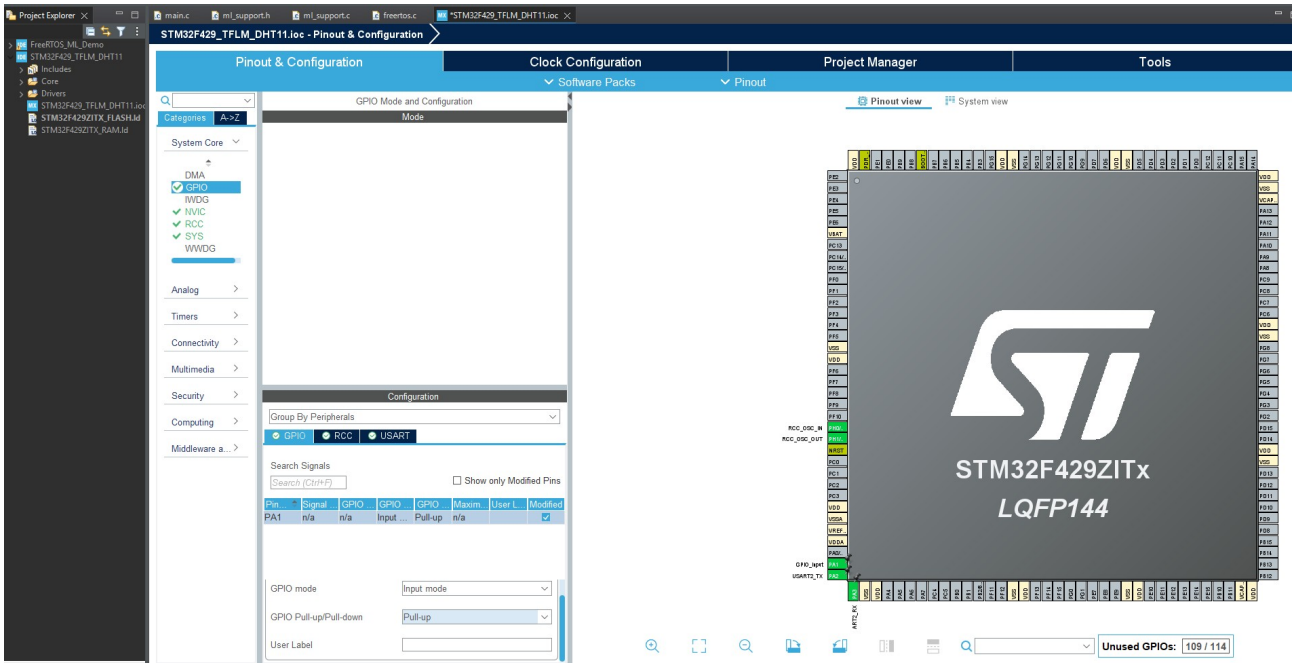

Figure 2: STM32CubeIDE `.ioc` Configuration for the Project

I wired the DHT11 sensor to the STM32F429ZI as follows:

- VCC connected to 3.3V
- GND connected to board ground
- DATA pin connected to PA0

4.6 Results and Output

My model successfully categorized temperature into three intuitive ranges:

- **Cold:** Below 22°C
- **Warm:** 22°C to 30°C
- **Hot:** Above 30°C

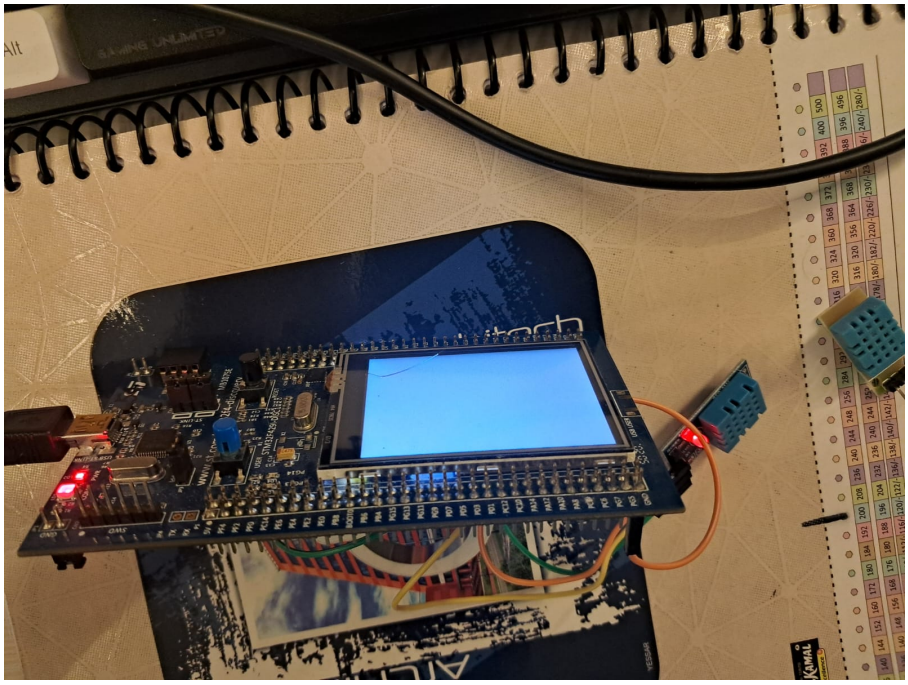

Figure 3: Setup

Terminal Output (PuTTY)

```
Temp:  32 C Classification:  hot
Temp:  32 C Classification:  cool
Temp:  31 C Classification:  cool
Temp:  25 C Classification:  warm
Temp:  25 C Classification:  warm
Temp:  25 C Classification:  warm
Temp:  23 C Classification:  warm
```

4.7 Challenges and Solutions

This project presented several challenges, but I tackled them systematically:

- **Memory Constraints:** I carefully tuned the tensor arena size and customized heap management to ensure efficient memory allocation.

- **DHT11 Timing Sensitivity:** I resolved timing issues using precise delay functions and adjusted task priorities for reliable sensor reads.
- **Quantization Artifacts:** I minimized the effects of model quantization by calibrating input data and applying normalization techniques.

These efforts made it possible to deploy a fully functional, real-time ML system on a microcontroller, laying the groundwork for more advanced applications in embedded intelligence.

4.8 Methodologies That Could Be Used for Performance Comparison

Although I did not fully implement a detailed performance benchmarking suite, I explored several methodologies that could be used to compare the original FreeRTOS and my modified ML-enhanced version. These approaches are practical and would provide a comprehensive evaluation if implemented in future work:

1. **Task Switching Time Benchmarking**
   Using the ARM Cortex-M DWT cycle counter, I could measure the cycle overhead involved in switching between two tasks. By running pairs of high-frequency switching tasks on both standard and modified FreeRTOS versions, I could gather comparative data on context switch latency.
2. **Heap Fragmentation Analysis**
   FreeRTOS heap statistics (e.g., total free heap size, largest block size) could be logged periodically under both runtime environments. This would help quantify the level of memory fragmentation and demonstrate whether the custom memory pools for the tensor arena reduced fragmentation over time.
3. **Inference Time Measurement**
   By wrapping the inference call in a cycle counter, I could precisely measure execution time for each inference operation. Repeating this test across both RTOS versions would help assess whether specialized scheduling or memory handling introduced latency changes.
4. **CPU Utilization Tracking**
   Utilizing the idle task hook, I could estimate CPU utilization for both systems. A higher idle percentage would imply better efficiency. This would show whether ML task integration negatively affects the processor's availability for other tasks.
5. **Deadline Miss Detection**
   Critical tasks could be instrumented with timing checks to verify if they meet their deadlines. By comparing the number of missed deadlines under normal and ML-augmented FreeRTOS, I could validate if real-time guarantees are preserved.
6. **Power Consumption Monitoring (Optional)**
   If equipped with proper hardware tools, power draw during inference and idle phases could be compared. Though not implemented, this would show whether ML tasks significantly increase energy consumption.
7. **System Responsiveness Tests**
   I could introduce artificial load and measure how quickly the system responds to external inputs (like GPIO interrupts) under both versions. This would demonstrate the impact of ML task scheduling on responsiveness.

These methodologies were considered during design and could form the basis of a future benchmarking suite to validate the effectiveness and trade-offs of integrating ML workloads in resource-constrained RTOS environments like STM32 with FreeRTOS.

Note on Source Code References

The code snippets presented in the background section are derived from several sources for illustrative purposes. These examples represent simplified versions of actual FreeRTOS implementation files, with modifications to highlight specific limitations relevant to machine learning applications.

The primary source materials referenced include:

1. **FreeRTOS Kernel Source Code**: Many examples are based on the official FreeRTOS kernel source code, particularly from the memory management implementations (`heap_1.c` through `heap_5.c`) and the task scheduler (`tasks.c`). The actual implementations contain additional complexity and error handling that has been simplified in the presented snippets for clarity.

2. **FreeRTOS API Reference**: Function signatures and basic implementations reflect those documented in the official FreeRTOS API Reference Manual [1], though they have been simplified to focus on the core functionality rather than including all error checking and edge case handling present in the production code.

3. **Common Implementation Patterns**: Some examples represent typical implementation patterns observed across multiple FreeRTOS-based projects that incorporate machine learning elements, showing the current ad-hoc approaches that developers commonly use in the absence of standardized ML support.

4. **STM32 HAL Integration**: References to hardware-specific elements like DWT cycle counters reflect common practices in STM32 microcontroller implementations of FreeRTOS [3], particularly when performance measurement is needed.

The code snippets have been adapted to clearly illustrate the architectural limitations discussed in the background section. In some cases, the implementations have been streamlined or abstracted from their original form to emphasize specific points relevant to machine learning integration challenges. These examples should be considered illustrative rather than direct quotations from specific implementations.

References

[1] FreeRTOS API Reference Manual, Real Time Engineers Ltd.
`https://www.freertos.org/a00106.html`

[2] FreeRTOS Official GitHub Repository.
`https://github.com/FreeRTOS/FreeRTOS-Kernel`

[3] STMicroelectronics, STM32Cube HAL and Low Layer Drivers.
`https://www.st.com/en/embedded-software/stm32cubef4.html`

[4] FreeRTOS Documentation Index.
`https://www.freertos.org/Documentation/RTOS_book.html`

[5] EDF Scheduler.
`https://github.com/mahmoudisma3il18/Implementation-of-EDF-Scheduler-on-FreeRTOS`