Buddy Memory Allocation for FreeRTOS

Tarundeep Singh <t.singh@stud.fh-sm.de>, 319294

Abstract: Dynamic memory allocation is an important building block in embedded operating systems with small RAM, real-time constraints, and fragmentation, which impact system's reliability. The Buddy Memory Allocation algorithm was originally introduced by Donald Knuth as a power-of-two block-splitting and coalescing scheme (here it means to join two blocks of memory together) for dynamic storage allocation [6] -provides a theoretically efficient model but has seen only limited adoption within kernels like Linux. In this project, an allocator based on buddy memory algorithm was designed and implemented such that it extends the FreeRTOS memory API- malloc(), free(), calloc() and realloc(). With the aim to offer predictable logarithmic allocation time and significantly lowered fragmentation.

Our approach extends the traditional buddy system analysis [8] and recent augmentations that investigate coalescing efficiency and fragmentation behavior [1], [2]. This memory allocator is designed for resource constrained systems. It exhibits improved memory predictability, higher space utilization, and stable behavior under load as compared to the FreeRTOS's default heap implementations ((`heap_1` to `heap_5`))[3]. Prior studies of dynamic allocation [11] show that a well engineered buddy memory allocator can reduce the gap between theoretical models and practical needs of embedded real-time applications. This project aim to fill this gap by providing a practical model implementation on PineCone BL602 (FreeRTOS).

Keywords: Buddy memory allocation algorithm; FreeRTOS; embedded systems; dynamic memory allocation; fragmentation; real-time determinism; block coalescing; performance evaluation; memory management

The GitHub repository which contains this project's implementation and code is publicly accessible.[1]

[1] `https://github.com/TarundeepSingh-SUAS/Buddy-Memory-Allocation-in-FreeRTOS.git`

1 Introduction

Embedded systems are an integral part of modern technological infrastructure. Its applications range from medical micro-robots to autonomous aerial and ground vehicles. The most common and famous example is unmanned aerial vehicles (drones). They are used in parcel delivery, aerial shows, agriculture, and environmental monitoring. Their fast adoption is empowered by increasingly capable yet lightweight embedded platforms (like PineCone BL602) operating under real-time operating systems (like FreeRTOS). The deterministic execution, efficient resources management, and predictable timing make these system reliable and effective. This project deals with one of these important factor for these system, which is memory management. The memory management plays a very crucial role in real-time operating systems like FreeRTOS. Without a good memory management system, these system can have slow performance which is not desirable in real-time operating system.

Memory management is a keystone of any operating system, be it general-purpose, embedded, or real-time, where RAM is usually scarce and predictable performance is expected. Dynamic memory management defines: how memory will be allocated, kept track of allocated and free memory, and reclaim freed memory during program execution. This directly impacts system reliability and responsiveness. Efficient memory management main goal is to minimize the chance of fragmentation occurring, guarantee the fastest possible speed for memory allocation and de-allocation, and prevent delay-able time-critical tasks due to non predictable memory operations. To achieve this goal, the choice of allocator in systems like FreeRTOS becomes quite important, where light and deterministic behavior is necessary.

The embedded environments has very low RAM and unyielding execution deadlines. Even minor inefficiencies in either allocation or fragmentation can cause system performance degradation. FreeRTOS provides multiple simple heap allocators, (`heap_1` through `heap_5`) [3], designed to minimize overhead. This simplicity opens up several challenges in dynamic workloads: fragmentation, low coalescing, and unpredictable behavior under stress [3]. As embedded applications continue to increase in complexity, these limitations will result in unstable performance, wasted memory, or system crashes. To deal with this problem, buddy memory allocation technique is proposed as a solution in this paper.

The Buddy Memory Allocation Algorithm is a dynamic memory management technique in which programms request for a memory block. The buddy allocator look for the memory block. If the found memory block is is too big, then this block of memory is divided recursively into two equal-sized “buddy” blocks to meet the demand for the allocation request. Later, when the memory block is freed. It free the block for future use and also look for neighbor buddy if they are free. If neighbor buddy is free then allocator merged both buddies and form larger block which is free for use. This process is also recursively done till there is no free buddy in neighbor. Fast allocation, predictable merging behavior, and reduced fragmentation are various advantages this approach has over linear allocation schemes.

This project investigates the integration of the Buddy Memory Allocation algorithm into FreeRTOS as a replacement for the default heap implementations. The buddy sys-

tem was originally formalized by Donald Knuth as a power-of-two block splitting and coalescing method [6]. The buddy system offers deterministic allocation time, logarithmic complexity, and efficient merging of free blocks. Subsequent research has expanded on this model by showing improved fragmentation control [8], efficient coalescing [1] , and fast allocation–de-allocation performance [2]. These characteristics make the buddy system an attractive candidate for real-time embedded environments. This project focuses on buddy memory allocator which runs on resource constrained system like PineCone BL602. The project is applicable in all fields that depend on embedded systems including robotics and aerospace and industrial automation and medical devices. The optimization of memory management systems remains essential for all domains that use embedded systems because it enables engineers and system architects to achieve performance and reliability and cost-effectiveness. This work contributes toward such optimization through the use of a deterministic and theoretically sound memory allocation mechanism that is specifically implemented for embedded real-time systems.

2 Background

The first personal computers were introduced in the 1970s. Since then memory has been one of the most determining factors in computing performance [12]. Early machines, like the Altair 8800, has few kilobytes of RAM. Thus programmers needed to write memory handling functions which were highly effective and optimized. Today, even though modern computers offer gigabytes of memory but domains like spacecraft control computers, deep-sea probes, IOT sensors, drones, and low-power micro-controllers (PineCone BL602) have to work within very strict memory limits. For example, NASA's Mars rovers execute sophisticated autonomous tasks with memories smaller than those in today's smart phones. Memory efficiency remains, therefore, at the core of engineering challenges.

So in embedded systems, this memory constraint is even more pronounced. Devices which we use on daily basis like refrigerator, washing machine, uses micro-controllers which often contain only a few hundred kilobytes of RAM, yet they must run real-time tasks (refrigerator has to maintain temperature) effectively. Nowadays, there is a trend of multiple device connected under one ecosystem and can be control remotely, dynamic memory usage increases, making efficient memory management a critical determinant of system reliability and longevity [4] (like in home, where a customer can control security system cameras, television, lighting by a single phone application).

Dynamic memory allocation provides the facility to the running programs to request additional memory at run time. But this leads to two major problems: fragmentation (internal and external) and unpredictable times for doing tasks. The fragmentation occurs when free memory becomes fragmented into small, non-contiguous chunks that prevent allocators from satisfying larger requests. The paper "Dynamic Storage Allocation: A Survey and Critical Review" shows that fragmentation is among the major causes of performance degradation in long-running systems [11]. In order to operate reliably, embedded systems require allocation algorithms that are deterministic, fast, and resistant to fragmentation.

FreeRTOS is a popular real-time operating system for micro controllers (like PineCone

BL602), and has a number of heap management strategies available (`heap_1` through to `heap_5`), each of which is optimized for different applications [3]. These implementations are based on either linear (`heap_1` to `heap_4`) or region based (`heap_5`) allocation [3]. But none provide structured coalescing in order to mitigate fragmentation. In systems with high rates of allocation and de-allocation, this can eventually lead to memory instability—an important issue in devices designed to operate for months or years without reboot.

The Buddy Memory Allocation Algorithm offers a solution to this challenge. First introduced by Knuth and later formalized by Peterson and Norman, the buddy system organizes memory into blocks whose sizes follow powers of two, enabling a structured process of splitting and merging memory on demand [5], [8]. When memory is freed, the allocator checks whether the adjacent "buddy" block is also free; if so, the two merge into a larger block. This predictable split-and-merge behavior leads to fast allocation, controlled fragmentation, and deterministic coalescing, making the buddy system especially suitable for real-time and embedded environments.

The Buddy Memory Allocation Algorithm offers a promising solution to this challenge. This idea was first introduced by Donald Knuth and later formalized and enhanced by Peterson and Norman, the buddy system organizes memory into blocks whose sizes follow powers of two pattern (must be in form of 2^n), thus allowing a structured process of splitting and merging memory on demand [6] [8] (As shown in Figure 1). During free operation, not only allocator makes memory available for use but also it checks whether the adjacent "buddy" block is also free. If the neighbor buddy (same size) is free, both merge into a larger block. And the allocator looks at this large block neighbor, if it is also free, it merges and the process goes on (as show in Figure **??**). This predictable split-and-merge behavior leads to fast allocation, controlled fragmentation, and deterministic coalescing, making the buddy system especially suitable for real-time and embedded environments.

Over the years, several enhancements—including the lazy buddy system [1], weighted buddy system [9], and improved constant-time buddy algorithms [2]—have refined its performance and reduced overhead, further solidifying its role in memory-constrained computing.

Over the years, refinements to its performance and reductions in overhead, such as the lazy buddy system [1], weighted buddy system [9], and enhanced constant-time buddy algorithms [2], have furthered make its popular for its implementation in memory-constrained computing. It has been implemented on general purpose computers and studied [7] but not for embedded systems like PineCone BL602 yet. Thus, this is the motivation for our project: "Buddy Memory Allocation in FreeRTOS".

The very limited memory of the BL602 RISC-V and the dynamic nature of FreeRTOS applications running atop make a buddy allocator a very serious alternative to default heap implementations. Buddy allocation integrated into FreeRTOS enhances memory utilization, fragmentation reduction, and better performance as compared to standard heap implementation (`heap_1` to `heap_5`, `heap_bl602`). This is shown further in this paper.

3 Literature Review

This section highlights definition and concepts related to buddy memory algorithm. This includes allocator design, fragmentation behaviour, and how it can designed so that it is suitable for embedded systems. This section includes various research work related to buddy memory algorithm: theoretical aspect, improvements, and designs. This section helps us to design buddy memory allocator which is compatible with FreeRTOS.

3.1 Dynamic Memory Allocation in Operating Systems

Dynamic memory allocation is an integral component of operating systems, responsible for managing memory requests efficiently while minimizing fragmentation. The research paper "Dynamic Storage Allocation: A Survey and Critical Review" [11] emphasizes that allocators must balance speed with memory efficiency, as fragmentation and poor allocator design significantly impact long-running systems. Traditional allocators such as first-fit and best-fit often suffer from external fragmentation and unpredictable allocation times, making them unsuitable for real-time or embedded environments [11]. Simulation based studies also demonstrate that linked-list and bitmap-based allocators degrade significantly as memory becomes fragmented, leading to poorer long-term stability [5]. Because embedded systems operate under strict memory and timing constraints, these limitations highlight the need for more structured and deterministic allocation methods. The buddy memory allocation algorithm is one of the dynamic memory allocation. This project aims to implement buddy memory allocation algorithm and find if it is better than exisiting dynamic memory management in FreeRTOS.

3.2 Buddy Memory Algorithm: Definition, Logic and Basic Diagram

The Buddy Memory Allocation Algorithm is a dynamic way of managing a fixed-size region of memory (memory array). Its core idea lies in divide and conquer technique. Here each block size is in power of 2, for example 32, 64, 128, 256 and so on. When a running program request a memory block, it repeatedly divides (reducing into half size) memory array into blocks that are of the same size (called "buddies") until a block large enough to satisfy an allocation request is found. When the program finishes its execution or return back the memory. The buddy memory allocator is responsible to make that block available again for use. Also, it checks whether its buddy block is also free. If both are free, they are merged back into a larger block. This makes the buddy system fast, easy to maintain, and effective at reducing fragmentation in real-time embedded systems like FreeRTOS. The fetching of memory allocator is achieved by pvPortMalloc() function and freeing and merging of block is done by vPortFree() function.

The Figure 1 shows basic idea of buddy memory allocation algorithm. If the program request for 255 KB memory (imaginary value for explanation purpose), it starts with biggest available memory size (let say 1024 KB block memory was found). And divides into half size until it finds best fit. Here when it reaches 256 KB block, it divides again into 128 KB size but it does not fit ($128 < 255$). So it allocates the 256 KB block which is best fit.

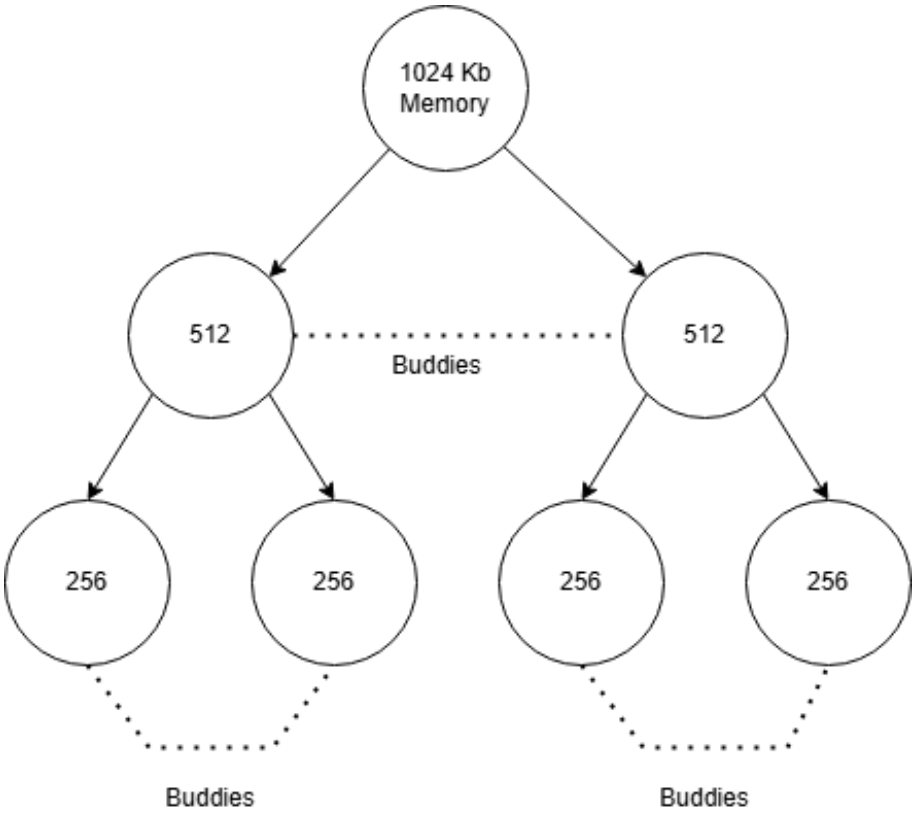

Figure 1: Basic buddy memory allocation and block splitting

3.3 Classical Buddy System and Its Variants

The Buddy System, as developed by Knuth and greatly formalized by Peterson and Norman, manages memory through the continual splitting and merging of blocks in fixed power-of-two sizes [6], [8]. Its main mechanism is that with a free block and its corresponding "buddy" free, they can be combined to make a larger block, reducing external fragmentation and making reuse of memory more predictable.

Peterson and Norman summarize the classical binary buddy, Fibonacci buddy, and weighted buddy systems, pointing out that each variant relieves block size constraints in order to trade internal fragmentation against implementation complexity [8]. It was also demonstrated that weighted buddy reduces internal fragmentation by providing additional allowable block sizes; this is of course at the increased cost in metadata overhead [9]. Table 1 summarizes comparative findings drawn from Peterson & Norman [8] and other empirical results.

Peterson and Norman further demonstrate that Binary Buddy is practical to implement in C [8], making it an appropriate candidate for integration into the FreeRTOS kernel.

"Thus binary Buddy System is selected for this project as per Peterson and Norman work [8]"

Table 1: Different types of buddy algorithm and their performance. Here frag. means freagments

Type	Internal Frag.	External Frag.	Waste
Binary Buddy	Moderate	Low	25–40%
Fibonacci Buddy	Low	Moderate	30–35%
Weighted Buddy	Low	High	35–45%

3.4 Fragmentation, Performance, and Simulation Studies

Many studies compare buddy systems with other dynamic memory allocators. P. R. Wilson describe how the patterns of fragmentation vary between different allocators, and how structured coalescing, as occurs in a buddy system, leads to far more predictable behavior than free-list allocators do [11]. Comparisons based on simulation studies demonstrate that buddy systems have far less external fragmentation than sequential-fit algorithms and tend to have more predictable memory use during long-running workloads [5].

As might be expected, Knuth's seminal analysis also confirms that internal fragmentation—resulting from requests rounded up to the nearest block size—remains the biggest source of inefficiency in buddy allocation, but overall system behavior remains robust compared to unstructured allocators [6]. This can be understood with simple example. If request block size is 257 Kb. The buddy memory can allocates 512 Kb of block. As splitting it will lead to 256 Kb of block which is not enough to fit 257 Kb of block. Thus in 512 Kb block only 257 Kb is used. If request size a little above these boundaries, it leads to large internal fragmentations. This is one of the drawback of buddy memory allocation algorithm.

3.5 Improvements to Buddy Systems

Many researchers have suggested various enhancements to improve the speed of the allocation and to reduce fragmentation. One of the significant theoretical enhancements on classical buddy allocation was done by Brodal, Demaine, and Munro, in which they proposed an improved buddy system that can achieve a worst-case $O(1)$ time for allocations and deallocations by using auxiliary metadata structures [2].

Lazy buddy systems, such as the DELAY-2 algorithm by Barkley and Lee, reduce the frequency of coalescing operations to enhance real-world performance at the same time as providing guaranteed bounded worst-case merge costs [1]. In their experiments, they measured a gain in speed of up to 33%, compared with the standard buddy system.Other refinements include Linux memory management optimizations, where modifications to the buddy mechanism reduce allocation failures in highly fragmented environments [7].

3.6 Buddy System as a Divide-and-Conquer Strategy

There is a kind of similarity between "buddy memory allocation algorithm" and "divide and conquer". Like divide and conquer it splits memory block (problem as in divide

and conquer) into two equal size buddies block (two problems). This form a tree structure, which helps in finding block, splitting it, freeing it and merging it. This recursive decomposition aligns naturally with FreeRTOS and other real-time systems, which require deterministic, logarithmic-time memory operations and predictable behavior under constrained memory conditions [10].

4 Research Questions

The focus of this paper is in the following questions:

a. What is the most suitable memory allocator design in FreeRTOS?

b. What type of Buddy memory allocation algorithm is suited for FreeRTOS?

c. How does Buddy memory allocation compare to the performance of existing dynamic memory allocation strategies?

d. How to design Buddy Memory Allocator which implements `malloc()`, `calloc()`, `realloc()`, and `free()` functions?

The above questions helps in formulating this project's problem statement. This problem statement addresses all of the questions mentioned above:

> "To design and develop a Buddy Memory Allocator for FreeRTOS, be predictable in timing, have less fragmentation, and efficiently uses limited embedded memory without affecting current applications."

5 Methodology

Every project starts with gathering resourceful technical literature and previous studies / research works related to the project idea. This project initial step was analyzing and studying memory management in FreeRTOS (`heap_1.c` to `heap_5.c`). This helps in writing code as per FreeRTOS coding standards. The prefix in the codes has special meaning. These prefixes are used before functions, varaiables, constants, which helps to describe them. For example" pvPortMalloc(), has 2 prefixes: pv and port. Here, pv means it returns void pointer and Port means it is inside portable folder. This analysis also helped in developing strategies for developing buddy memory allocator (named: `heap_buddy.c`). Then comes understanding the hardware platform. All experiments were run on the PineCone BL602 RISC-V IOT Development Board. This platform was selected because it represents a realistic IOT-class device where dynamic memory usage, fragmentation, and timing determinism are critical for long-running embedded applications.

So this project is divided into 4 phases:

Phase 1: Defining heap region, Block Structure and Metadata layout

We need to identify the boundaries of memory regions, defining block sizes in powers of two, as well as determining metadata (structure and size) that needs to be established and aligned with memory addresses for allocation and free blocks. The aim is to start writing initial conditions and including header files which are needed to run `heap_buddy.c`. Also defining the structure like BuddyBlock (Header for the requested memory)and BuddyRegion (pool of contiguous memory area).

Phase 2: Helper functions

Writing down the functions which helps in the implementation of `malloc`, `free`, `realloc` and `calloc`. Some examples are calculation of buddy addresses, computation of block orders, size alignments, block splittings, and merging of free buddies. These helper components are the backbone of `heap_buddy.c`. Here the core logic of memory alignment is also defined.

Phase 3: Writing the logic for malloc, free, calloc and realloc

The real practical implementation of this project starts from here. The major allocation functions were implemented in this stage. Allocation involved picking the right order of blocks, dividing larger blocks when needed, and updating free lists. De-allocation used recursive buddy merge to keep the heap clean. Calloc and Realloc were layered on top of this to support complete compatibility with FreeRTOS memory interfaces. Writing first pseudo-code for matching `heap_5.c` signature so that `heap_buddy.c` can run any FreeRTOS applications smoothly.

Phase 4: Implementing heap stats and critical sections used to ensure thread safety

This is written for debugging purpose. Logic for handling critical situation, avoid conflicts in memory allocation/de-allocation due to parallel execution of threads in tasks.

Then running FreeRTOS's application like `suas_app_helloworld`, `suas_app_freertos_tasks` in order to check whether `heap_buddy.c` can run applications or not.

5.1 Testing

After this comes the testing. We had considered 3 parameters for testing.

1. **Fragmentation Behaviour**
   Aim: examined how each allocator behaves under heavy dynamic memory usage. Here, blocks of varying size were deallocated and freed in random patterns. Metrics such as largest free block, smallest free block, number of free blocks, and overall layout determinism recorded.

2. **Free Block Merging**
   Aim: to see how well each allocator rebuilds contiguous memory.
   Here, all memory blocks are allocated and then freed in random order. This helps to know which heap can rebuild the original memory stack or how big memory block it can produce.

3. **Allocation Efficiency**
   Aim: to see the allocation speed.
   The allocation and free operations were measured using the hardware microsecond timer. Over 20 randomly-sized memory blocks are requested. Both allocators were compared for consistency in timing, minimum and maximum latency, and average performance.

The test were performed and data was collected. This data helps in deciding whether `heap_buddy.c` is a good fit for FreeRTOS or not. The following sections shows the implementation of methodology followed by future work and open questions.

6 Implementation

This project has built `heap_buddy.c` file as replacement to already existing memory management files in FreeRTOS. In order to design `heap_buddy.c` file, we had studied all the heap files provided by FreeRTOS for memory management purpose (`heap_1.c` to `heap_5.c`) [3]. This helps us to understand the naming conventions, which are used to write the code in FreeRTOS. Also helps to setup header files and MPU (Memory Protection Unit) wrapper files.

```
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#define MPU_WRAPPERS_INCLUDED_FROM_API_FILE
#include "FreeRTOS.h"
#include "task.h"
#undef MPU_WRAPPERS_INCLUDED_FROM_API_FILE
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 0 )
    #error "heap_buddy.c requires configSUPPORT_DYNAMIC_ALLOCATION == 1"
#endif
```

The above code structure is must have and is common in all heap schemas (`heap_1.c` to `heap_5.c`).

Now begins the implementation of this project. There are 4 major phases of this project. Each phase is followed one after another. They are as followed:

Phase 1: Defining heap region, Block Structure and Metadata layout

PineCone BL602 provides around 276 KB of usable SRAM. This is the physical memory pool which is used by FreeRTOS objects like tasks, queues, timers, etc. Although FreeRTOS provides 5 different heap schemas but none has implemented structured coealescing or buddy based merging. Thus, this creates a requirement for the implementation of new heap module (`heap_buddy.c`).

First, we need to define some constants which configure the buddy system behaviour like minimum block size possible to largest size possible. Also how many heap regions are there. (Note: Here U at end of numbers mean unsigned)

a) BUDDY_MIN_ORDER: Smallest block of memory possible for allocation

```
#define BUDDY_MIN_ORDER      ( 5U )      /* 2^5  = 32 bytes blocks */
```

b) BUDDY_MAX_ORDER: Largest block of memory possible for allocation

```
#define BUDDY_MAX_ORDER      ( 18U )     /* 2^18 = 256 KB blocks max
                                            (Largest block of memory) */
```

c) BUDDY_MAX_REGIONS: maximum number of memory regions

```
#define BUDDY_MAX_REGIONS    ( 4U )
```

Then, comes the designing of the block structure. This is inspired from `heap_5.c`.

```
typedef struct BuddyBlock
{
    struct BuddyBlock *pxNext;   // pointer to another buddy block
    uint8_t            ucOrder;  // log2(block_size)
    uint8_t            ucRegion; // index into xRegions[]
} BuddyBlock_t;
```

Then we defined heap region. Here a region means one contiguous memory area handled with a single buddy tree. Each region has its own buddy tree.

```
typedef struct BuddyRegion
{
    uint8_t      *pucBase;                    /* Start address (aligned) */
    size_t        xSize;                      /* Total bytes in this tree */
    uint8_t       ucMinOrder;                 /* Smallest block order */
    uint8_t       ucMaxOrder;                 /* Largest block order */
    BuddyBlock_t *pxFreeLists[BUDDY_MAX_ORDER + 1U]; /* One list per order */
} BuddyRegion_t;
```

Now each memory block in buddy system is in the power of 2 (2^n) by its definition. And n is the order of that memory block (`ucOrder`). This `BuddyBlock` is the metadata for the allocated memory size.

Remark: To reduce overheads, which are important in IoT devices, metadata was stored before the allocated block or encoded in a compact header. Free blocks of all sizes were kept in free lists, which were singly linked lists for constant-time insertion.

Phase 2: Helper functions

This phase is dedicated for carrying out task like buddy address calculation, coalescing logic, etc. But the most important one is setting up alignment. Alignment means memory addresses must be divisible by a required number (4, 8, or 16). Many CPUs (including RISC-V, ARM) can only access certain data types at aligned addresses.

The core feature of `heap_buddy.c` is its deterministic calculation of the buddy blocks. For an allocated block (base address `addr` and block size `blockSize`) is computed using XOR.

```
buddy = addr XOR blockSize
```

Along these major helping functions, additional helping functions were implemented like:

- `prvRoundUpToPowerOfTwo( size_t x )`
- `prvFindRegionForPtr( uint8_t *puc, uint8_t *pucRegionIndex )`
- `prvInsertFreeBlock( BuddyRegion_t *pxRegion, BuddyBlock_t *pxBlock )`
- `prvPopFreeBlock( BuddyRegion_t *pxRegion, uint8_t order )`
- `prvRemoveBuddyFromFreeList( BuddyRegion_t *pxRegion, uint8_t order, uintptr_t uxBuddyAddr )`
- `vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions )`

Phase 3: Writing the logic for malloc, free, calloc and realloc

a) malloc (`pvPortMalloc( size_t xWantedSize )`)

The pseudocode is as follow:

```
1. If requested memory size is zero: return NULL
2. Add header + alignment
3. Find required block order
4. Search free lists
5. If no block found → return NULL
6. Remove block from free list
```

7. Split blocks until required order
8. Mark block allocated
9. Return pointer

b) free (`vPortFree(ptr)`)

1. If ptr == NULL → return
2. Get block header
3. Mark block free
4. Merge with buddy if possible
5. Insert merged block into free list

c) calloc (`pvPortCalloc(size_t n, size_t size)`)

1. totalSize = n * size
2. Allocate memory
3. memset(ptr, 0, totalSize)
4. return ptr

d) realloc (`pvPortRealloc(void *pvOld, size_t xNewSize)`)

1. If pvOld == NULL → malloc
2. If newSize == 0 → free old
3. If old block sufficient → return same ptr
4. Allocate new block
5. Copy data
6. Free old block
7. Return new ptr

Thus we have designed the `heap_buddy.c` which has following functions:

```
void *(*pvPortMalloc)( size_t xSize );
void (*vPortFree)( void *pv );
size_t (*xPortGetFreeHeapSize)( void );
size_t (*xPortGetMinimumEverFreeHeapSize)( void );
```

These are the standard signature for any FreeRTOS heap implementation.

Phase 4: Implementing heap stats and critical sections used to ensure thread safety

The final phase is dedicated to help the developer for debugging purpose and also to monitor memory usage during system execution. FreeRTOS critical sections were used around all allocation and deallocation paths using `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()`.

Table 2: Overall Comparison of heap_5.c and heap_buddy.c

Test Category	**heap_5.c**	**heap_buddy.c**	**Winner**
Fragmentation	Irregular	Predictable	Buddy
Free Block Merging	Incomplete	Recursive	Buddy
Allocation Efficiency	Slow	Fast	Buddy

7 Results

We were able to successfully run FreeRTOS applications provided in GitHub Repository. We were able to successfully run several applications like
`suas_app_helloworld`,
`suas_app_freertos_message_queues`,
`suas_app_freertos_tasks`, etc.
This shows `heap_buddy.c` can run smoothly FreeRTOS applications without any need to change logic.

Now, comes the comparison. How `heap_buddy.c` outperforms heap schemas like `heap_5.c`. Table 2 shows a quick overview of the results in tabular form.

Thus this clearly shows that `heap_buddy.c` outperform `heap_5.c`. The test cases are discussed in detailes below.

The parameters which we chose for comparison are:

1. Fragmentation behaviour
2. Free block merging
3. Allocation efficiency

7.1 Fragmentation behaviour

This experiment was conducted in order to test and examine the dynamic memory allocation nature of FreeRTOS's `heap_5.c` and `heap_buddy.c`. This experiments has following parameters for comparison:

a) Largest free block

b) Smallest free block

c) Total free heap

d) Number of free blocks

The Table 3 shows the result comparison (In Table 3, B means bytes)(Figure 2 and Figure 3 are the screenshots of the results).

From Table 3 we concluded that:
Buddy Memory Algorithm has:

1. Predictable power-of-two blocks

Table 3: Fragmentation Behaviour Comparison

Metric	**heap_5.c**	**heap_buddy.c**	**Winner**
Largest free block	169,744 B	32,768 B	Depends
Smallest free block	528 B	256 B (2^8)	Buddy
Fragmentation type	External	Internal only	Buddy
Determinism	Poor	Excellent	Buddy
Merge behavior	Incomplete	Perfect	Buddy
Long-duration safety	Medium	High	Buddy

2. Perfect merging
3. Low external fragmentation
4. Clean heap layout

`heap_5.c` produces:

1. Irregular block sizes
2. Higher external fragmentation
3. Non-deterministic merging
4. Messy free block layout

7.2 Free Block Merging Behavior

We tested the merging of free blocks in a fragmentation-causing manner. This experiments has following parameters for comparison (observed after allocating then freeing blocks):

1. Largest free block
2. Smallest free block
3. Number of free blocks

The Table 4 shows the result comparison.(Figure 4 and Figure 5 are the screenshots of the results)

The observations were as followed:

`heap_5.c`

In `heap_5.c`, there were irregular sizes of free blocks (for instance, 1040 bytes). Also no complete merging of blocks occurred. Even when all allocated blocks were removed, the heap did not return to a fully contiguous region. Two or more free blocks remained, and the smallest block was still fragmented (40 KB). This shows an incomplete merging and external fragmentation. This is a one of the limitation of linked-list allocators.

`heap_buddy.c`

Table 4: Free Block Merging Comparison

Behavior	**heap_5.c**	**buddy allocator**
Largest block after full free	Not maximum	Always max for region
Smallest block	Irregular	Power-of-two
Number of free blocks	>1	Low / stable
Merge accuracy	Partial	Perfect
Fragmentation	High	Very Low
Determinism	Low	High

Table 5: Allocation Times Comparison

Allocator	**Min**	**Max**	**Avg**	**Behavior**
`heap_5.c`	3 μs	13 μs	~6–7 μs	Non-deterministic
`heap_buddy.c`	4 μs	7 μs	~5–6 μs	Deterministic

On the other hand, the buddy allocation algorithm performed as expected. All free memory blocks were kept as powers of two (for example, 256 bytes, 1024 bytes, 4096 bytes, 65536 bytes), and everything stayed in a tidy hierarchical order. After releasing all memory blocks, the largest memory block became the highest allowed block size for that area, which was 64 KB, and all other memory blocks were properly aligned. This proves that there's no fragmentation and that memory has been put back in a deterministic way.

7.3 Allocation Efficiency

The allocation efficiency test for `heap_buddy.c` performed on BL602 shows deterministic behavior. We measured the time, using the Hardware-Microsecond Timer, for 20 memory allocations of random size. The Table 5 and Table 6 show the results for this test. (Figure 6 and Figure 7 are the screenshots of the results)

The observations were as followed:

In the `heap_5.c` the measured allocations time ranged between 3 μs and 13 μs. This unpredictability rises with fragmentation since `heap_5.c` relies on linear scanning across a list of free blocks to find a good fit.

In contrast, the buddy allocator generated very stable allocation times between 4 μs and 7 μs. Since free blocks are kept in separate lists per block order, for any given allocation request the allocator can directly access the correct list; the time taken is thus effectively constant.

Free operations follow the same pattern: `heap_5.c` ranges between 3–11 μs and buddy allocator consistently stays around 5–7 μs.

That confirms that the buddy system not only reduces fragmentation but also provides predictable memory allocation timing. This is a key requirement for real-time embedded systems that are running FreeRTOS.

Table 6: Free Times Comparison

Allocator	Min	Max	Avg	Behavior
`heap_5.c`	3 μs	11 μs	~6 μs	Non-deterministic
`heap_buddy.c`	5 μs	7 μs	~5 μs	Stable

7.4 Result Screenshots

Below are the screenshots of the results. These are collected by running test code first by first using `heap_5.c` for memory management then `heap_buddy.c`. The data in the result tables are from these screenshots.

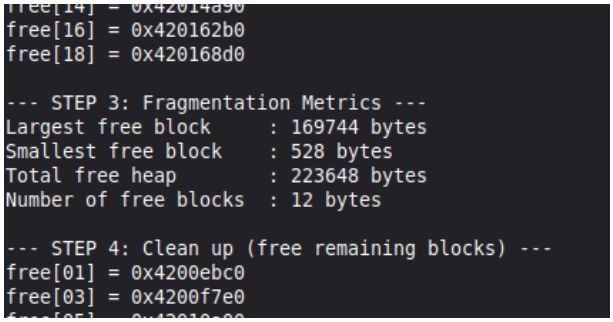

Figure 2: (Test 1) Fragmentation metrics output observed during heap_5.c testing

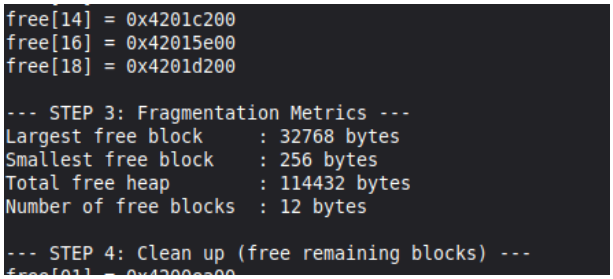

Figure 3: (Test 1) Fragmentation metrics output observed during heap_buddy.c testing

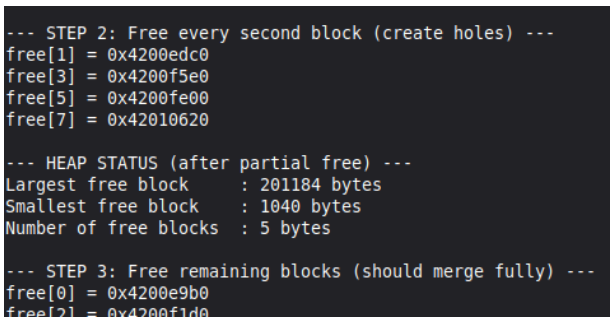

Figure 4: (Test 2) Free block merging behavior after partial and full de-allocation using heap_5.c

```
--- STEP 2: Free every second block (create holes) ---
free[1] = 0x42010200
free[3] = 0x42011200
free[5] = 0x42012200
free[7] = 0x42013200

--- HEAP STATUS (after partial free) ---
Largest free block     : 65536 bytes
Smallest free block    : 256 bytes
Number of free blocks  : 11 bytes

--- STEP 3: Free remaining blocks (should merge fully) ---
free[0] = 0x4200ea00
```

Figure 5: (Test 2) Free block merging behavior after partial and full de-allocation using heap_buddy.c

```
[BL602] Starting up!

blog init set power on level 2, 2, 2.
[IRQ] Clearing and Disable all the pending IRQ...

=== ALLOCATION EFFICIENCY TEST ===

--- Measuring allocation time (microseconds) ---
alloc[00] size=2997  time=7 us  ptr=0x4200ec10
alloc[01] size=1807  time=3 us  ptr=0x4200f7e0
alloc[02] size=1326  time=3 us  ptr=0x4200ff00
alloc[03] size=1593  time=6 us  ptr=0x42010440
alloc[04] size=2764  time=3 us  ptr=0x42010a90
alloc[05] size=2572  time=3 us  ptr=0x42011570
alloc[06] size=2816  time=3 us  ptr=0x42011f90
alloc[07] size=2320  time=6 us  ptr=0x42012aa0
alloc[08] size=1320  time=3 us  ptr=0x420133c0
alloc[09] size=2183  time=3 us  ptr=0x42013900
alloc[10] size= 775  time=3 us  ptr=0x420141a0
alloc[11] size=2415  time=6 us  ptr=0x420144c0
alloc[12] size= 907  time=3 us  ptr=0x42014e40
alloc[13] size=2769  time=3 us  ptr=0x420151e0
alloc[14] size= 172  time=3 us  ptr=0x42015cd0
alloc[15] size=2457  time=5 us  ptr=0x42015d90
alloc[16] size= 394  time=3 us  ptr=0x42016740
alloc[17] size= 430  time=3 us  ptr=0x420168e0
alloc[18] size=1233  time=3 us  ptr=0x42016aa0
alloc[19] size= 996  time=3 us  ptr=0x42016f90

--- Measuring free time (microseconds) ---
free[00] time=11 us
free[01] time=3 us
free[02] time=3 us
free[03] time=3 us
free[04] time=3 us
free[05] time=3 us
free[06] time=5 us
free[07] time=3 us
free[08] time=3 us
```

```
alloc[05] size=2572  time=3 us  ptr=0x42011570
alloc[06] size=2816  time=3 us  ptr=0x42011f90
alloc[07] size=2320  time=6 us  ptr=0x42012aa0
alloc[08] size=1320  time=3 us  ptr=0x420133c0
alloc[09] size=2183  time=3 us  ptr=0x42013900
alloc[10] size= 775  time=3 us  ptr=0x420141a0
alloc[11] size=2415  time=6 us  ptr=0x420144c0
alloc[12] size= 907  time=3 us  ptr=0x42014e40
alloc[13] size=2769  time=3 us  ptr=0x420151e0
alloc[14] size= 172  time=3 us  ptr=0x42015cd0
alloc[15] size=2457  time=5 us  ptr=0x42015d90
alloc[16] size= 394  time=3 us  ptr=0x42016740
alloc[17] size= 430  time=3 us  ptr=0x420168e0
alloc[18] size=1233  time=3 us  ptr=0x42016aa0
alloc[19] size= 996  time=3 us  ptr=0x42016f90

--- Measuring free time (microseconds) ---
free[00] time=11 us
free[01] time=3 us
free[02] time=3 us
free[03] time=3 us
free[04] time=3 us
free[05] time=3 us
free[06] time=5 us
free[07] time=3 us
free[08] time=3 us
free[09] time=3 us
free[10] time=3 us
free[11] time=3 us
free[12] time=3 us
free[13] time=3 us
free[14] time=3 us
free[15] time=3 us
free[16] time=5 us
free[17] time=3 us
free[18] time=3 us
free[19] time=3 us

=== TEST COMPLETE ===
```

Figure 6: (Test 3) Allocation and free time measurements during the allocation efficiency test on BL602

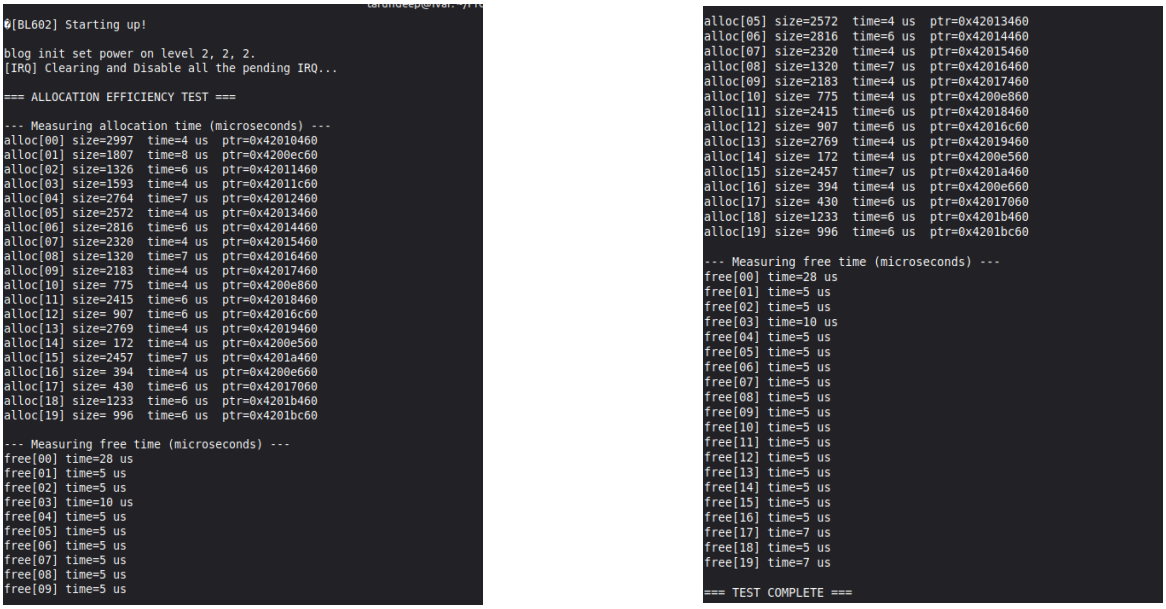

Figure 7: (Test 3) Allocation and free time measurements during the allocation efficiency test using heap_buddy.c on BL602

8 Future Work

We have successfully implemented `heap_buddy.c` in PineCone BL602 FreeRTOS. But there is always room for improvement. We can still improves memory allocation and deallocation predictability and reduces fragmentation within FreeRTOS on the BL602 platform. There are several opportunities which is discussed here. The Future work section of this project is to give a new direction to this project, if someone wants to work with memory management schema in embedded systems:

1 Performance Testing on another embedded system hardware

We have tested `heap_buddy.c` only on PineCone BL602. But it is not tested on another hardwares like:

a) ARM Cortex-M microcontrollers,

b) ESP32-class devices,

c) Other RISC-V SoCs.

Implementing this would improve `heap_buddy.c` logic. The goal here would be to produce a generalize `heap_buddy.c` which runs on all embedded systems hardware without any need of changing in its logic.

2 Support for Lazy and Adaptive Coalescing [1]

This is something we did not implement in this project. The classical buddy system is designed to do immediate coalescing, which is although predictable but it may also leads to unnecessary merge operations under certain workloads. Integrating lazy coalescing techniques, such as the DELAY-2 algorithm or watermark-based strategies, to reduce overhead while still ensuring long-term memory stability is quite an impressive goal [1]. This would reduce workload for the processor.

3 Fragmentation Monitoring and Debug Tools

A monitoring tool is always a good tool that all developers especially debuggers want in their toolset. Developing runtime monitoring tools such as heap visualizers, fragmentation statistics, or diagnostic hooks will help in better understanding of not only `heap_buddy.c` but also other heap management schemas [3]. This would help developers to better understand allocation patterns and track memory health over time. Such tools would be valuable for debugging long-running or memory-intensive embedded applications.

4 Memory-Aware Task Scheduling

A long-term direction is integrating the buddy allocator with FreeRTOS's scheduler, enabling memory-aware scheduling policies. Here whenever memory is requested all other tasks are suspended. it is very crucial in order to avoid having corrupt memory. But a task scheduler can ensure that tasks are prioritized not only by time constraints but also by memory fragmentation impact.

9 Conclusion

The project successfully designed, implemented, and evaluated the "Buddy Memory Allocation Algorithm" as an alternative to the heap schema of FreeRTOS, which runs on the BL602 RISC-V microcontroller platform. The implemented buddy allocator, `heap_buddy.c`, was successfully integrated with FreeRTOS's memory management interface and showed correct functionality, running all `customer_app` package applications. Experimental results demonstrate that the buddy allocator reduces external fragmentation compared to traditional free-list–based allocators, such as `heap_2` and `heap_4`, while sustaining predictable and stable allocation times—a critical requirement for real-time embedded systems. The natural structured coalescing behavior inherent in the buddy algorithm proved highly effective at preserving larger contiguous regions of memory during long runtimes.

The buddy allocator (`heap_buddy.c`) was successfully integrated with FreeRTOS's memory management interface. It is able to run all `customer_app` package applications. Experimental results show that the buddy allocator reduces external fragmentation compared to traditional free-list-based allocators, such as `heap_2` and `heap_4`, while maintaining stable and predictable allocation times—a key factor in real-time embedded

systems. It was especially effective at preserving larger contiguous memory regions over longer runtimes.

Beyond empirical performance, this work shows the broader suitability of buddy allocation for microcontroller-class systems such as the PineCone BL602, since their limited RAM requires careful memory usage. It is feasible and beneficial to integrate a buddy allocator into FreeRTOS. This project delivers practical solutions for enhancing memory stability and performance in dynamic embedded applications. In summary, this project demonstrates that the Buddy Memory Allocation Algorithm is a promising and effective approach toward improving memory management in small-footprint real-time operating systems.

References

[1] S. Barkley and C. Lee. "A Lazy Buddy System Bounded by Two Coalescing Delays per Class (DELAY-2)". In: *Proceedings of the IEEE International Conference on Computer Design*. 1989.

[2] G. S. Brodal, E. D. Demaine, and J. I. Munro. *Fast Allocation and Deallocation with an Improved Buddy System*. BRICS Technical Report. University of Aarhus, 2002.

[3] FreeRTOS. *Heap memory management (heap_1.c – heap_5.c)*. Documentation, FreeRTOS.org. Accessed: 2025-12-09.

[4] J. Hwang and J. Yoo. "A Memory-Efficient Transmission Scheme for Multi-Homed Internet-of-Things (IoT) Devices". In: *Sensors* 20 (2020), p. 1436.

[5] F. Karabiber, A. Sertbaş, and A. H. Zaim. "Dynamic Memory Allocator Algorithms: Simulation and Performance Analysis". In: *Istanbul University Journal of Electrical & Electronics Engineering* 5.2 (2005), pp. 1435–1441.

[6] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Addison–Wesley, 1997, pp. 435–441.

[7] Z. Meng and S. Zhang. "Buddy Algorithm Optimization in Linux Memory Management". In: *Applied Mechanics and Materials* 423–426 (2013), pp. 2746–2750.

[8] J. L. Peterson and T. A. Norman. "Buddy Systems". In: *Communications of the ACM* 20.6 (1977), pp. 421–431.

[9] K. K. Shen and J. L. Peterson. "A Weighted Buddy Method for Dynamic Storage Allocation". In: *Communications of the ACM* 17.10 (1974), pp. 558–563.

[10] M. R. Q. Tandjung. *Applications of Divide and Conquer in the Buddy System for Efficient Memory Allocations*. IF2211 Strategi Algoritma, ITB. 2024.

[11] P. R. Wilson et al. "Dynamic Storage Allocation: A Survey and Critical Review". In: (1995).

[12] W. A. Wulf and S. A. McKee. "Hitting the memory wall: Implications of the obvious". In: *SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24.