422 Project Documentation

By: Mohammed Mohiuddin, Rayyan Karim, Nolan Dela Rosa

Section I: Introduction

Assembly programming plays a vital role in system-level programming, particularly in embedded systems and operating systems. It provides direct control over hardware and system resources, which is crucial for performance and reliability in constrained environments. This paper documents the implementation of critical memory management and signal-handling functions in assembly. The functions discussed, including `_bzero`, `_strncpy`, `_malloc`, `_free`, `_alarm`, and `_signal`, demonstrate fundamental low-level programming concepts while addressing challenges like memory safety and efficient resource management.

Section II: Overview of Functions

Table 1.1 provides a summary of the memory management functions implemented in this program.

Table 1.1: Memory Management Functions

Function	Description
_bzero	**parameters** = (string *s, int n) Initializes a memory block to zero to prevent undefined behavior from residual data in memory. The function uses loops to efficiently zero memory addresses, ensuring all locations within the specified range are cleared. A common use case is to clear buffers before reusing them to eliminate the risk of corruption or data leakage. **helper methods/branches:** **_bzero_loop:** basically a branch used to loop until the size reaches N and zeroes out loops at given location s. **_bzero_finished:** branch that gets reached once the loop finishes.
_strncpy	**parameters** = (char* dest, char* src, int size) Safely copies a string from a source to a destination buffer, adhering to a specified size limit. By avoiding buffer overflows, the program ensures

	robust memory management, even when handling strings of varying lengths. This function is invaluable in test cases where strings must be manipulated or stored without compromising memory integrity. **helper methods/branches:** **_strncpy_loop:** basically a branch that loops until one of two conditions gets met. Either size reaches 0, or we reach the end of the string. We made two checks to account for the scenario if either or happens or if a size is given that exceeds the length of the given strings **_strncpy_finished:** branch to finish once the loop ends.
_malloc	**parameters = (int size)** Dynamically allocates memory during runtime using system calls such as SYS_MALLOC. When this method gets called it relays it call to SVC Handler with a number in R7 that then goes to _systemcall_table_jump() in svc. There _kalloc is called in heap.s which contains the actual logic and code for this method.
_kalloc	**parameters = (int size)** This method is used to set up the registers and the MCB by loading the start and end and saving the necessary registers before relaying its call to the recursive allocator _ralloc. The reason _ralloc is in a different tab is because it has its own relative branches and helpers that are used.
_ralloc	**parameters = (int size, int left_mcb_addr, int right_mcb_addr)** The function attempts to allocate memory within a specified range, defined by the left and right MCB (Memory Control Block) addresses. It recursively searches for a suitable memory block that can accommodate the requested size, ensuring efficient memory management. It checks the availability of memory blocks, splits the address space, and makes recursive calls to search the left and right halves of the memory space. It prevents memory over-allocation and fragmentation by marking used blocks and ensuring contiguous allocation.

	helper methods/branches: **recursive_left_half:** Recursively attempts to allocate memory in the left half of the address space. **recursive_right_half:** Recursively attempts to allocate memory in the right half of the address space if the left half fails. **Return_zero:** Handles the case when a suitable memory block is not found in this case we set R6 = 0; **str_half_heap:** Marks the right MCB and stores the size of half the heap. **ralloc_finished:** Concludes the _ralloc function, returning the allocated memory address or zero. We only come to this branch if allocation was successful.
_free	**parameters = (void* addr)** Releases dynamically allocated memory back to the system using SYS_FREE. Proper deallocation is crucial to prevent memory leaks and ensure resource availability for future operations. When this method gets called it relays its call to SVC Handler with a number in R7 that then goes to _systemcall_table_jump() in svc. There _kfree is called in heap.s which contains the actual logic and code for this method.
_kfree	**parameters: = (void *ptr)** The _kfree function frees a memory block pointed to by ptr. It validates the address to ensure it is within the valid heap range, calculates the corresponding MCB address, and calls _rfree to perform the deallocation and possible merging of memory blocks. **helper_methods/branches:** **_rfree:** recursive helper to free out memory blocks **kfree_INVALID:** Handles invalid cases where the pointer is out of the heap range. Puts 0 in R0 to indicate failure **kfree_finished:** Concludes the _kfree function and restores the link register.

_rfree	**parameters: (int mcb_addr)** The _rfree function deallocates a memory block by marking the Memory Control Block (MCB) at the given address as free. It checks for adjacent (buddy) blocks that can be merged to reduce fragmentation and improve memory management. The function operates recursively to ensure that all possible merges are performed. **helper_methods/branches:** **check_buddy_below:** Checks if the buddy block below the current MCB is free and merges if possible. Recursively calls _rfree to continue the process. **rfree_INVALID:** Handles invalid cases where the MCB address is out of bounds or merging is not possible. sets the return status to 0, indicating failure. **rfree_finished:** Concludes the _rfree function and restores the link register. We only branch here if deallocation is successful
syscall_table_init	**parameters: none** This method is triggered in the reset_handler, where its call initializes the relative memory spaces for the program to run and calls the functions like kalloc, kfree, timer_start, and signal_handler properly.
syscall_table_jump	**parameters: none** When the SVC Handler routine gets triggered via the bit move to R7 and the SVC call in stdlib, the call from the handler gets relayed to the syscall_table_jump which then determines where and which functions to call. It does some basic operations with registers to determine the address (stored in r8) as well as offset using r7. **helper_methods/branches:** **free_method:** if r7 has the number corresponding to free we will branch here to call the appropriate function kfree in heap

	malloc_method: if r7 has the number corresponding to malloc we will branch here to call the appropriate function kalloc in heap **alarm_method:** if r7 has the number corresponding to alarm we will branch here to call the appropriate function _timer_start in timer **signal_method:** if r7 has the number corresponding to alarm we will branch here to call the appropriate function signal_handler in timer **finish:** This function will be branched after the relative function has been done running and returns back to its original location in svc. From then registers are restored and the program moves on.
heap_init	**parameters: none** **This method is basically responsible for initializing the heap memory space to be used for allocation and deallocation** **helper_methods/branches:** **init_loop:** Loop to initialize from a starting address to an ending address with 00s to indicate empty memory space. **init_finished:** branch here once the loop has finished initializing.
timer_init	**parameters: none** Initializes the timer by setting the systick control register to stop the systick timer, and storing the maximum reload value in the systick reload register.

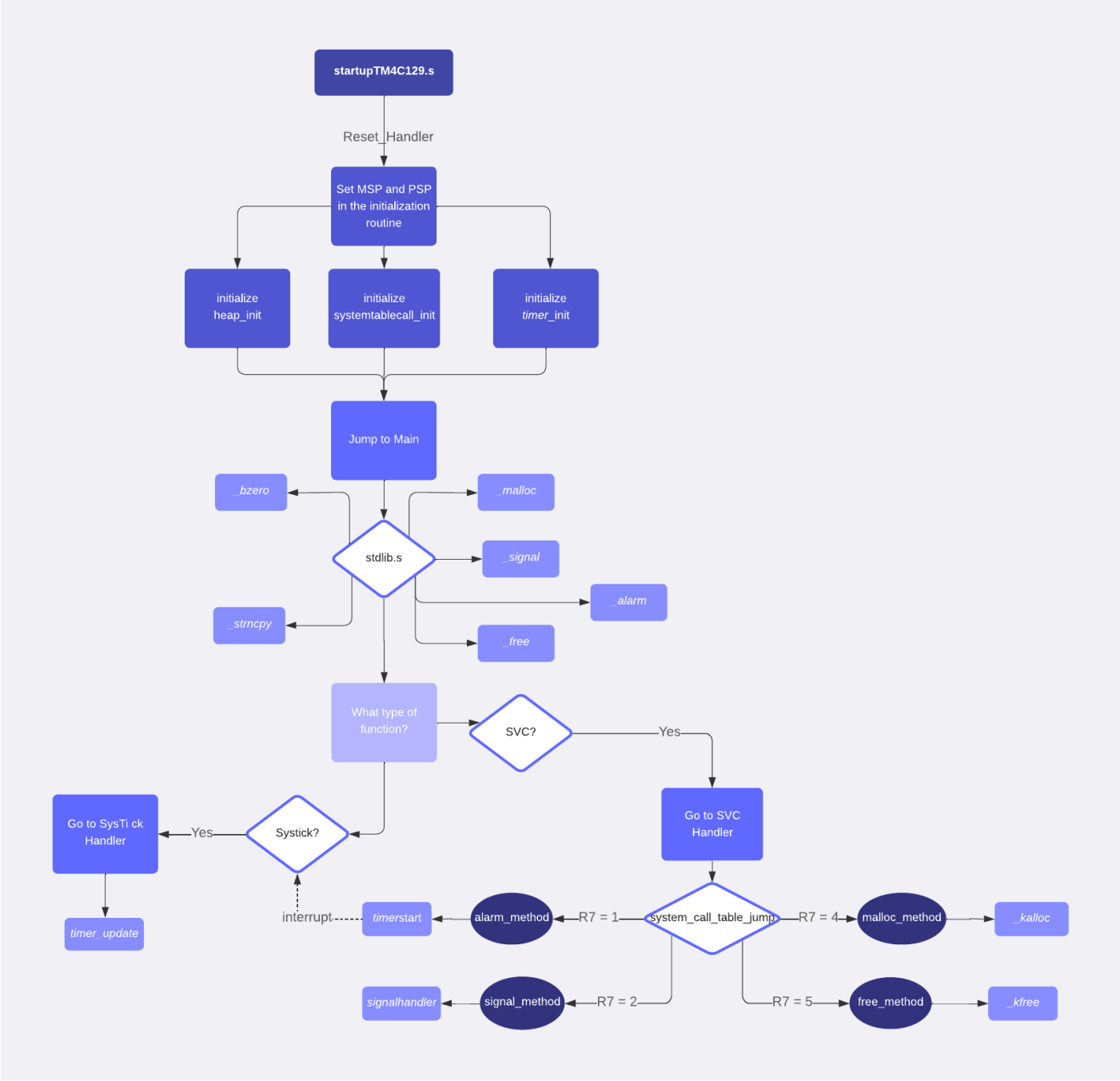
Table 1.2 provides a summary of the timer functions implemented in this program.

Table 1.2: Timer Functions

Function	Description
_alarm	**parameters:** int seconds The alarm function sets a timer that will signal after a specified number of seconds. It handles the timing mechanism by relaying the seconds argument from the main function to _timer_start in timer.s. The function also retrieves and returns the previous timer value, starts the SysTick timer, and manages the timer's status and value. returns Number of seconds remaining for a previously set alarm, or zero if none. Helper functions: **helper_methods/branches:** **timer_start:** Starts the systick timer, sets the new countdown value specified by a parameter seconds, and clears the systick current value register. ***timer_update:** this is a function that basically is used not inside of the alarm but is technically a part of the alarm. When a SysTick interrupt happens, it will get triggered by the Systick Handler in startup to update the timer. It will update the seconds left counter, check if it reaches zero, ends the program if it hasn't, or stops the timer if it has. **timer_update_done:** just finished the program by moving pc to lr **see Section IV: Challenges/Limitations*
_signal_handler	**parameters**: Signal number (signum) and a pointer to the handler (handler). Allows the registration of custom handlers for various signals. The function dynamically updates internal signal mappings, enabling the system to execute user-defined behaviors in response to specific signals like SIGALARM. This flexibility is crucial in creating responsive and adaptive applications. **returns**: The pointer to the previously installed handler for the given signal.

Figure 1.1: Final Program Design

Please note the following diagram that outlines the entire program flow from start to finish. This program will end at the end of driver.c and will continue as long as functions are being called. This is a general outline of how the driver is working and the various pathways that may be taken from the start and those paths are triggered.



Section III: Implementation Checklist

Our final project is fully functional, incorporating all required components and meeting all specified objectives including:

- **startup_tm4c129.s**
  - ☑ Reset_Handler
  - ☑ SVC_Handler
  - ☑ SysTick_Handler

- **stdlib.s**
  - ☑ _bzero( )
  - ☑ _strncpy( )
  - ☑ _malloc( )
  - ☑ _free( )
  - ☑ _alarm( )
  - ☑ _signal( )

- **svc.s**
  - ☑ _systemcall_table_init( )
  - ☑ _systemcall_table_jump( )

- **heap.s**
  - ☑ _kinit( )
  - ☑ _kalloc( )
  - ☑ _ralloc( )
  - ☑ _kfree( )
  - ☑ _rfree( )

- **timer.s**
  - ☑ _timer_init( )
  - ☑ _timer_start( )
  - ☐ _timer_update( ) *(see Section IV: Challenges/Limitations)*
  - ☑ _signal_handler( )

All execution snapshots of driver are attached as a folder in the .zip file final submission folder

Section IV: Challenges/Limitations

One notable challenge we encountered during the implementation involved the function _time_update(), which currently triggers a Memory Access Prevention error. This issue exposes an inherent limitation within our _timer_update() implementation regarding linking and proper memory access management, and we will elaborate on its causes, potential implications, and mitigation strategies.

What Currently Works

So far our timer_update() is able to properly get relayed a call by a systick interrupt, read the address at the SECONDS_LEFT, and is able to decrement or branch based off whether its 0 or not as specified by the final pdf.

Limitations

The problem occurs when it invokes the call to the relative USR_Handler, in this case, which is _malloc. and then throws a memory access management issue saying read permission denied.

This issue doesn't really affect the rest of the flow of the program and is only inherently related to the timer itself as well was it the functions of timer. Everything outside of this function including (the user handler function invoked outside of this content i.e malloc) works as normal.

Mitigation Strategies

Due to the small and intricate nature of this issue as well as the specific conditions required to produce it, we weren't able to track the root cause. However, some potential starting points may include, checking that the processor is in the correct mode, checking the linker script, and debugging the entire flow to determine why the function is accessing the relative address and/or why the error is being thrown.