Final Report

When starting this project, our group initially set up the Keil development environment and established a GitHub repository, emphasizing the importance of organization in software development. We dove into the complexities of low-level programming by translating 'heap.c' into assembly language, specifically creating 'heap.s'. This task was fundamental in understanding the memory management at a low level.

A significant part of our project involved implementing the Memory Control Block ('_kinit') and developing recursive techniques for memory allocation and freeing in '_kalloc' and '_kfree'. These functions required a profound understanding of memory management strategies and the ability to execute complex algorithms in assembly language.

We also took on the challenge of translating high-level standard library functions, such as 'bzero' and 'strncpy', into 'stdlib.s'. This process required setting up system call handlers for various functions, showing the complexity of system architecture and the interplay between different components.

Our project further looked into managing the SysTick timer and handling interrupts, testing our skills in precision and timing, crucial for system-level programming. This was particularly obvious in our handling of SIG_ALRM signals, which was important in understanding system responses and interruptions.

Managing memory addresses and system variables was a task that demanded precision, efficiency, and a methodical approach. Additionally, adapting to the Keil C compiler's requirements for the Master Stack Pointer (MSP) and Process Stack Pointer (PSP) mapping highlighted the importance of adaptability in software development.

Reflecting on this project, we realize that this project was more than just coding in assembly language; it was an eye opener on system architecture and low-level programming. Each step presented its own set of challenges and learnings, significantly contributing to our growth as software developers and enhancing our problem-solving skills, attention to detail, and technical understanding of the ARM architecture and system programming nuances.

Summary

In our project, we initially set up the Keil environment and a GitHub repository, emphasizing organization in software development. We delved into low-level programming by translating 'heap.c' into 'heap.s', a critical step in understanding memory management. Implementing the Memory Control Block ('_kinit') and developing recursive functions for memory allocation ('_kalloc') and freeing ('_kfree') required deep knowledge of system memory dynamics. We also translated high-level standard library functions into 'stdlib.s', setting up system call handlers and illustrating the complexity of system architecture. Managing the SysTick timer and handling interrupts honed our precision and timing skills, essential in system-level programming. The project further involved managing memory addresses and system variables, and adapting to the Keil C compiler's requirements for stack pointer mapping. This journey was more than coding; it was an immersive experience into system architecture and low-level programming, enhancing our problem-solving skills and technical understanding of ARM architecture.

Narratives:

Implemented:

startup_tm4c129.s:

Reset_Handler: Initializes the system upon reset.

SVC_Handler: Handles Supervisor Calls (SVCs)

SysTick_Handler: Manages system tick interrupts.

driver_keil.c: Serves as the driver program, interfacing and testing the implemented functions.

stdlib.s:

_bzero(): Implements the bzero function, which sets memory blocks to zero.

_strncpy(): Implements the strncpy function, providing string copy functionality.

_malloc(): Implements memory allocation functionality.

_free(): Handles memory deallocation, freeing up previously allocated memory blocks.

_alarm(): Implements alarm functionality, although it was not fully tested.

_signal(): Manages signal handling.

svc.s:

_systemcall_table_init(): Initializes the system call table.

_systemcall_table_jump(): Facilitates the jumping to appropriate system call handlers.

heap.s:

_kinit(): Initializes the memory control blocks for memory management.

_kalloc(): Allocates memory blocks as per the request.

_ralloc(): A recursive helper function for allocating memory spaces.

_kfree(): Frees up allocated memory blocks.

_rfree(): A recursive helper function for freeing up memory spaces.

timer.s:

Missing:

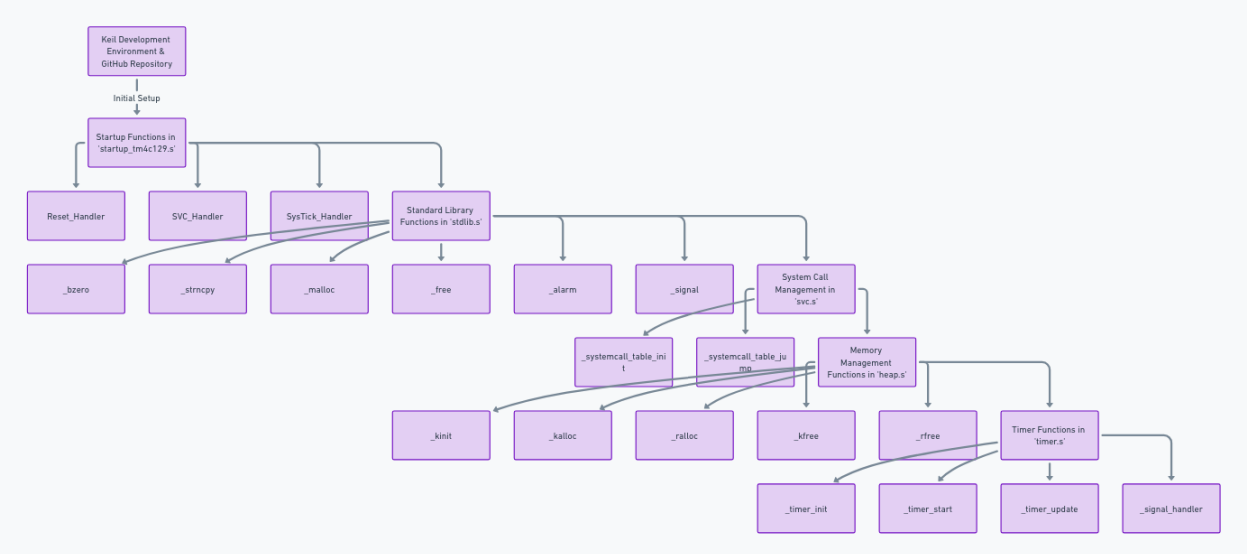
_timer_init(): Initializes the timer.

_timer_start(): Starts the timer, crucial for tracking time intervals.

_timer_update(): Updates the timer, managing the countdown or up-count functionalities.

_signal_handler(): Handles the signals, particularly for time-related events.

Diagrams:



EXTRA CREDIT

_strlen

Functionality: Calculates the length of a string until it encounters a null terminator.

Description: The _strlen function iterates over each character of the provided string, incrementing a counter until it reaches the null terminator. This function is essential in scenarios where the length of a string is required, and it does not modify the original string.

_memsets:

Functionality: Fills a block of memory with a specified character.

Description: _memsets is a custom implementation of the standard memset function. It sets a specified number of bytes in a memory block to a given character. This function is particularly useful for initializing or resetting a memory area with a specific byte value.

_toupper:

Functionality: Converts a lowercase character to uppercase. It has no effect on characters that are not lowercase letters.

Description: The _toupper function transforms a single character from lowercase to uppercase. If the character is not a lowercase letter, the function returns the character unchanged. This function is commonly used in text processing and formatting.

_strcmp:

Functionality: Compares two null-terminated strings and returns an integer based on the lexicographical comparison.

Description: _strcmp performs a character-by-character comparison of two strings. The function returns a value indicating the relationship between the strings: equal (0), the first string is lexicographically less than the second (-1), or the first string is greater (1). It's a fundamental function for sorting and searching operations in text data.

_strcat:

Functionality: Concatenates two strings, appending the second string to the first.

Description: The _strcat function appends the second string to the end of the first string. It overwrites the null terminator at the end of the first string and then adds a new null terminator after the last character of the concatenated string. This function is crucial for combining strings in various applications.