Basil OS: Programmer's Manual

Function	Type and Usage (Optional)	Purpose and algorithm	Return Values	Parameters / Struct Members
int itoa(int value, char *buf, int width, int base)	Function, String formatting	This function converts an integer into a non-NULL terminated string. Can convert to any base and digit count. Programmer must supply the number of digits: if too many digits are given the string will include leading zeros, if too little are given the most significant digits will be cut off. The method iterates through each char in the output string starting at the least significant digit, setting each to the ASCII value for the input / base ^ power, then modulo base.	The function returns the unprinted most significant digits of the input value, or value / (base ^ width). This value can be used in another itoa() call to convert more of the input value to a string.	**Int value**: the integer to be converted to a string. **char *buf:** the string to store the result in. **int width:** The number of digits in the output string. **int base**: The base of the output string. Bases higher than 10 will use letters from the English alphabet for higher value digits, and bases higher than 36 will use ASCII characters following 'Z' for higher value digits.
int serial_poll(device dev, char *buffer, size_t len);	Function, I/O	This function takes in a string, read from a serial port. It stores the string in buffer and tracks the bytes read in len The method uses a temp char c to accept each keystroke using inb(). It then uses strcmp to pair the correct input with an appropriate function. The method tracks the character positions	This function returns an int of the number of bytes read, or a negative if reading failed	**Device dev:** the defined serial port to read data from **char *buffer:** a char array where the read-in string will be stored **Size_t len:** the number of bytes written, stored as type size_t

		using a buffer, and modifies the buffer to match the typical functionality of typing.		
int serial_out_post(device dev, const char *buffer, size_t len, const unsigned int extra_whitespace)	Function, I/O	This function writes a buffer to a specified serial port without editing the cursor; it also adds extra whitespace to clear undesired old characters This method's algorithm executes by using serial_out to output the desired inputs and uses two for loops, using outb to print the appropriate whitespace, and serial_out (LEFT) to shift the cursor to the correct position.	This function returns an int of the number of bytes written	**Device dev:** The serial port to send the desired output **Const Char *buffer:** The desired string to be outputted to the port **Size_t len:** The desired number of bytes to write **Const char extra_whitespace:** The desired number of whitespaces to add to the end
int serial_buffer_backspace(device dev, char *buffer, unsigned int * const pos_ptr, unsigned int * const buffer_filled_ptr)	Function, I/O	This function performs the backspace functionality akin to that of a normal keyboard This method subtracts from the position pointer and uses serial out (LEFT) to shift the cursor. A for loop is used which subtracts from the buffer to shift the existing characters backwards. Finally, serial_out_post is called to correct the whitespace.	This function returns a -1 if the backspace is at the start of a buffer, 0 otherwise	**Device dev:** The serial port to output the backspace **Char *buffer:** The buffer to perform a backspace in **Unsigned int * const pos_ptr:** A pointer to the position in the buffer, which allows the backspace to update cursor **Unsigned int * const buffer_filled_ptr:** A pointer to the buffer capacity is stored, used for error handling in edge cases
int serial_buffer_delete(device dev, char	Function, I/O	This function performs the delete functionality akin to that of a	This function returns a -1 if the backspace	**Device dev:** The serial port to output the backspace **Char *buffer:** The buffer to

*buffer, unsigned int * const pos_ptr, unsigned int * const buffer_filled_ptr)		normal keyboard This method uses a for loop which subtracts from the buffer to shift the existing characters backwards by 1. Finally, serial_out_post is called to correct the whitespace.	is at the end of a buffer, 0 otherwise	perform a delete in **Unsigned int * const pos_ptr:** A pointer to the position in the buffer, which allows the delete to update cursor **Unsigned int * const buffer_filled_ptr:** A pointer to the buffer capacity is stored, used for error handling in edge cases
int strtolower(char* s)	Function, String formatting	This function converts all ASCII uppercase characters to their lowercase counterparts This method uses a for loop to iterate over the given string, and calls the tolower_inplace function on each, adding each successful execution.	This function returns an int of the number of characters converted	**Char * s:** The desired string to be converted to lowercase
int isdigit(char c)	Function, string formatting	This function confirms that a given character is an ASCII digit, 0~9. The function just does a simple comparison check to see if the character is between ASCII ‘0’ and ‘9’.	1 if the character is an ASCII digit, 0 otherwise.	**Char c:** The desired char to check
char tolower(char c)	Function, String formatting	This function converts a singular ASCII char to its lowercase counterpart The given character is checked to see if its ASCII value is between ‘A’ and ‘Z’, all the uppercase characters. If so, the character + ‘a’ - ‘A’ is	This function returns the singular converted char	**Char c:** The desired char to be converted to lowercase

		returned. Otherwise, the character unchanged is returned.		
int tolower_inplace(char *c)	Function String formatting	This function converts a singular ASCII char to its lowercase counterpart, doing so to the value at the given pointer. The given character is checked to see if its ASCII value is between ‘A’ and ‘Z’, all the uppercase characters. If so, the character is set to itself + ‘a’ - ‘A’, and 1 is returned. Otherwise, the character isn’t changed and 0 is returned.	This function returns an int of 1 if the char was converted, 0 otherwise	**Char *c:** The desired in-place char to be converted to lowercase
void comhand(void)	function	The main function of the OS, starting the command-line interface The function enters an infinite while loop where it queries for user commands, calling the appropriate command handling functions depending on it.	none	none
Void get_rtc(rtc_time_t *t)	Function, RTC handling	This function retrieves a new struct containing RTC elements, converting the raw BCD values to binary, to perform the date/time get commands	none	**Rtc_time_t *t:** a pointer to the RTC struct to be populated with date and time values

Void set_rtc(uint8_t day, uint8_t month, uint8_t year, uint8_t hour, uint8_t minute, uint8_t second)	Function, RTC handling	This function writes to the RTC registers to perform the date/time set commands	none	**Uint8_t: day, month, etc.:** The binary value of the associated time variable to be written to, i.e. day will write to the RTC's day register
static uint8_t read_rtc_register(uint8_t reg)	Function, RTC handling	This helper function performs the byte reading and writing necessary for retrieving data from the RTC	This function returns the associated CMOS data from the RTC	**Uint8_t reg:** the identifier byte to be called (seconds, minutes, etc) to the CMOS address for reading.
static void write_rtc_register(uint8_t reg, uint8_t value)	Function, RTC handling	This helper function performs the byte writing necessary for writing data to the RTC	none	**Uint8_t reg:** the identifier byte to be called (seconds, minutes, etc) to the CMOS address for writing. **Uint8_t value:** the value, in BCD, to be written to the RTC
static uint8_t bcd_to_bin(uint8_t val)	Function, RTC handling	This helper function performs the math to convert from BCD to binary for use in date/time commands	This function returns the newly converted binary	**Uint8_t val:** the value, in BCD, to be converted to binary
static uint8_t bin_to_bcd(uint8_t val)	Function, RTC handling	This helper function performs the math to convert from binary to BCD for use in date/time commands	This function returns the newly converted BCD	**Uint8_t val:** the value, in binary, to be converted to BCD
typedef struct { uint8_t second; uint8_t minute; uint8_t hour; uint8_t day; uint8_t month; uint8_t year; } rtc_time_t;	Struct, RTC handling	This struct stores the various outputs associated with the RTC for use in the date and time commands	This struct can be called on with the variable rtc_time_t	The associated uint8_t elements in this struct correspond with their nomenclature, i.e. seconds corresponds to the RTC value for seconds
R2 ADDITIONS				

void cmd_pcb(char** toks, int token_count);	Void function, command dispatcher for pcb-related operations	Interprets the second token of a pcb command, and then invokes the corresponding pcb operation. The function first checks if ‘toks[1]’ exists, if it does not it will print an error message instructing the user on how to use it correctly. If it does exist then it will use a series of else if conditions and ‘strcmp’ to compare ‘toks[1]’ against the command strings, once it finds the one that matches it will call the correlating function.	none	**char** toks**: An array of C-string tokens representing the command and arguments **Int token_count**: An integer that specifies the total number of tokens in the ‘toks’ array
void cmd_pcb_create(char ** toks, int token_count);	Void function, this function is called when a user uses the command to create a new PCB **This function has been removed as of R3.**	This function creates a new PCB. First it checks to make sure that there is 5 or more tokens if its less then an error is printed, then it will get the name from ‘toks[2]’, type from ‘toks[3]’ and priority from ‘toks[4]’ and convert to an integer using ‘atoi’. Next it will validate that the name is unique, if not it will print an error message, next it will check the type to see if it is ‘user’ or ‘system’ and set the ‘type_int’ flag to 1 for ‘user’ and 0 for ‘system’. It will also validate that the	none	**char** toks**: an array of strings representing the command line input. Expected tokens are, pcb, create, pcb name,pcb type, pcb priority as a string. **int token_count**: the total number of tokens provided in the toks array.

		priority is between 0 and 9 if not it will print an error. If it passes all validations a success message is printed and ‘create_pcb(name, type_int, priority)’ is called to create the new PCB		
void cmd_pcb_show(char** toks, int token_count);	Void function, processes the ‘pcb show’ command	This function displays the details of a specified PCB. First, it checks to ensure that there are exactly 3 tokens; if not, an error message is printed indicating the proper formatting ‘pcb show [name]’. Then, it retrieves the PCB's name from ‘toks[2]’ and calls ‘pcb_find(name)’ to locate the PCB. If no PCB is found, it prints "PCB not found." Otherwise, it prints a header with column names ("NAME", "CLASS", "STATE", "SUSPENDED_STATE", "PRIORITY") and calls show_pcb(name) to display the PCB's details.	none	**char** toks**: An array of strings representing the command line input. Expected tokens are ‘pcb, show, pcb name’ **int token_count:** integer representing the total number of tokens provided.
void cmd_pcb_delete(char ** toks, int token_count);	Void function, handles the deletion of a specified pcb	This function deletes a specified PCB. First, it ensures that there are exactly 3 tokens in the input; if not, it prints an error message instructing the user on the proper format (pcb delete [name]). Next, it	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, delete, pcb name’ **int token_count:** integer representing the number of tokens provided.

		retrieves the PCB's name from toks[2] and uses pcb_find(name) to locate the PCB. If the PCB is not found, it prints "PCB not found." If the PCB is found, the function checks its class type: if it is a user PCB (PCB_USER), it calls delete_pcb(name) to delete it and prints a success message; if the PCB is a system PCB, it prints an error stating that system PCBs cannot be deleted by users.		
void cmd_pcb_block(char** toks, int token_count); **This function is now removed as of R6**	Void function, processes the ‘pcb block’ command	This function blocks a specified PCB. First, it checks to ensure that there are exactly 3 tokens; if not, it prints an error message with the correct format (pcb block [name]). Then, it retrieves the PCB's name from toks[2] and calls pcb_find(name) to locate the PCB. If no PCB is found, it prints "PCB not found." Otherwise, it calls block_pcb(name) to block the PCB and prints a success message indicating that the PCB has been blocked successfully.	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, block, pcb name’ **int token_count:** integer representing the number of tokens provided.

void cmd_pcb_unblock(char** toks, int token_count); **This function is now removed as of R6**	Void function, handles the ‘pcb unblock’ command	This function unblocks a specified PCB. First, it checks to ensure that there are exactly 3 tokens; if not, it prints an error message indicating the correct format (pcb unblock [name]). Then, it retrieves the PCB's name from toks[2] and uses pcb_find(name) to locate the PCB in the system. If the PCB is not found, it prints "PCB not found." Otherwise, it calls unblock_pcb(name) to unblock the PCB and prints a success message confirming that the PCB has been unblocked successfully.	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, unblock, pcb name’ **int token_count:** integer representing the number of tokens provided.
void cmd_pcb_suspend(char** toks, int token_count);	Void function, handles the ‘pcb suspend’ command	This function suspends a specified PCB. First, it checks to ensure that there are exactly 3 tokens; if not, an error message is printed with the correct format (pcb suspend [name]). Then, it retrieves the PCB's name from toks[2] and uses pcb_find(name) to locate the PCB. If the PCB is not found, it prints "PCB not found." If the PCB is found and its class type is PCB_USER, it calls suspend_pcb(name)	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, suspend, pcb name’ **int token_count:** integer representing the number of tokens provided.

		to suspend the PCB and prints a success message. If the PCB is a system PCB, it prints an error message stating that users cannot suspend system PCBs.		
void cmd_pcb_resume(char** toks, int token_count);	Void function, handles the ‘pcb resume’ command	This function resumes a specified PCB. First, it checks to ensure that there are exactly 3 tokens; if not, it prints an error message instructing the user on the correct format (pcb resume [name]). Then, it retrieves the PCB's name from toks[2] and calls pcb_find(name) to locate the PCB. If the PCB is not found, it prints "PCB not found." Otherwise, it calls resume_pcb(name) to resume the PCB and prints a success message indicating that the PCB has been resumed successfully.	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, resume, pcb name’ **int token_count:** integer representing the number of tokens provided.
void cmd_pcb_priority(char** toks, int token_count);	Void function, handles the ‘pcb resume’ command	This function updates the priority of a given PCB. First, it checks to ensure there are exactly 4 tokens; if not, it prints an error message with the correct format (pcb priority [name] [new priority]). Next, it retrieves the PCB's name from toks[2] and	none	**char** toks**: Array of strings representing command line tokens, expected tokens are ‘ pcb, priority, pcb name, new priority value as a string’ **int token_count:** integer representing the number of tokens provided.

		converts the new priority value from toks[3] to an integer using atoi(). Then, it locates the PCB using pcb_find(name). If the PCB is not found, it prints "PCB not found." Otherwise, it verifies that the new priority is within the valid range (0 to 9). If the priority is valid, it calls set_pcb_priority(name , priority) to update the PCB's priority and prints a success message; if not, it prints an error message indicating that the priority value must be between 0 and 9.		
void cmd_pcb_showready( );	Void function, displays a list of all PCB's in a ready state	This function displays the list of ready PCBs. First, it prints a table header (using the PCB_TABLE_HEAD constant) to label the columns. Then, it calls show_ready() to output the details of each PCB that is in the ready state.	none	none
void cmd_pcb_showblock ed();	Void function, displays a list of all PCB's in a blocked state	This function displays the list of blocked PCBs. First, it prints a table header using the PCB_TABLE_HEAD constant to label the output columns. Next, it calls show_blocked() to output the details of	none	none

		each PCB that is in the blocked state.		
void cmd_pcb_showall();	Void function, displays a list of all PCB's	This function displays the complete list of PCBs. First, it prints a table header using the constant PCB_TABLE_HEAD to label the columns. Then, it calls the helper function show_all() to output the details of every PCB managed by the system.	none	none
int is_leap_year(uint8_t year);	Integer function, indicates whether or no the given year is a leap year	This function determines whether a given year is a leap year. First, it checks if the year is divisible by 4. Then, it ensures that if the year is divisible by 100, it must also be divisible by 400. If the year satisfies these conditions, the function returns a non-zero value (true) indicating that it is a leap year; otherwise, it returns 0 (false).	Type: int Returns 0 if given year is not a leap year. Returns a non zero value if the year provided is a leap year.	**uint8_year**: year to be evaluated, provided as a 8-bit unsigned integer
int month_length(uint8_t month, uint8_t year);	Integer function, returns an integer representing the number of days in a month for a specified year	This function calculates the number of days in a specified month of a given year. First, it checks if the month is February (denoted by the constant FEB). If so, it calls is_leap_year(year) to determine if the year	Type: int Returns number of days in a given month and year	**uint8_t month**: predefined constant (FEB, APR, etc.) **uint8_t year**: year to be evaluated, provided as an 8-bit unsigned integer

		is a leap year. If the year is not a leap year, it returns 28; otherwise, it returns 29. If the month is not February, it checks if the month is one that has 30 days (April, June, September, or November, represented by the constants APR, JUN, SEP, and NOV). If it matches one of these, the function returns 30. For all other cases (months with 31 days), it returns 31.		
int imin(int a, int b);	Integer function, returns an int representing the smaller of two ints	This function finds and returns the minimum of two given integers. First, it compares the two integers, a and b. If a is less than b, it returns a; otherwise, it returns b. This simple conditional check ensures that the function outputs the smallest value among the two inputs.	Type: int Returns the variable that is less than the other.	**Int a**: integer to compare **Int b**: integer to compare
int get_i_width(int value, int base);	Integer function, Returns an int representing the number of digits needed to represent an integer in a specified base	This function calculates the "width" of an integer, meaning the number of digits required to represent it in a specified base. First, it checks if the input value is 0; if so, it returns 1, as zero is represented with a single digit. If the value is greater than 0, the function initializes a counter (width) to zero. It then	Type: int Returns 1 if the value is 0 Otherwise it will return the count of digits required to represent the value in the specified base	**int value**: a non negative integer whose width is to be determined **int base:** The numeral base in which to represent the number

		enters a loop where it repeatedly divides the value by the base, incrementing the width counter on each iteration. This loop continues until value becomes 0, with the counter reflecting the number of digits needed. Finally, the function returns the computed width.		
const char *strchr(const char* str, char chr);	const char * function, returns a pointer to character within a string	This function searches the input string str for the first occurrence of the character chr. First, it initializes a pointer a to iterate over the string starting from str. It then enters a loop where it checks each character (using a[0]) against the target character chr. If a match is found, the function immediately returns the pointer to the matching character within the string. If the loop reaches the end of the string without finding chr, the function returns NULL to indicate that the character was not found.	Type: const char * Returns a point to the first occurrence of ‘chr’ with the string ‘str’ if found else it will return NULL	**const char *str**: a NUL-terminated string in which the search is performed. **char chr:** the character to search for within the string.
const char *strpbrk(const char* str, const char* key);	const char * function, returns a pointer to a character within a string	This function locates the first occurrence in the string str of any character that appears in the string key. First, it iterates over each character in str using a pointer (a).	Type: const char * returns a pointer to the first occurrence of any character from ‘key’ found in ‘str’. If no	**const char *str**: the NUL-terminated string in which the search is performed. **const char *key**: The NUL-terminated string containing the set of characters to search for within ‘str’

		For each character in str, it then iterates over each character in key using another pointer (b). If a character in str matches any character in key, the function immediately returns the pointer to that character in str. If no matching character is found after traversing the entire string, the function returns NULL.	match is found it will return ‘NULL’	
int isnumber(const char* str)	Function, string formatting	Checks that a given string is some sort of integer value. First, the function checks the first character of the string to see if its plus or minus sign. If so, it checks the next character to see if it's a digit with isdigit. If not, it checks the first character again to see if it's a digit with isdigit.	1 if string is number, 0 otherwise.	**const char* str**: string to check for numberness
void cmd_time(char** toks, int token_count);	Void function that is a dispatcher for the time related commands	This function handles time commands. First, it checks that a second argument is provided. If the second token is missing, it prints an error message instructing the user on the proper usage of the time command. If the second token is "get", the function calls cmd_time_get() to retrieve the current	none	**char** toks**: Array of string representing the tokens from the command line input. Expected tokens are ‘time, get or set, others’ **int token_count**: Integer that represents the total number of tokens provided by the array.

		time. If the token is "set", it calls cmd_time_set(toks, token_count) to update the time. For any other value of the second token, it prints an error message indicating an invalid argument and provides guidance to use "help time" for proper usage.		
void cmd_time_get();	Void function, retrieves and displays the current time	This function retrieves the current time from the RTC and prints it to the console. First, it declares a variable now of type rtc_time_t to store the time. It then calls get_rtc(&now) to populate now with the current time values. Finally, it prints the hour, minute, and second using printf in a formatted manner (%2d:%2d:%2d), ensuring the output is in a readable "hh:mm:ss" format.	none	none
void cmd_time_set(char** toks, int token_count);	Void function, sets the time based on user input	This function sets the current system time. First, it checks to ensure that at least 5 tokens are provided; if not, it prints an error message indicating the correct format (time as "HH MM SS" in military time). If the correct number of tokens is available, it converts the tokens for hour, minute, and	none	**char** toks**: Array of strings representing the tokens from the command line input. Expected tokens 'time, set, hour value, minute value, second value' **int token_count**: the total number of tokens provided in the 'toks' array. must have at least 5

		second (tokens 2, 3, and 4) from strings to integers using atoi(). Next, it validates the time values to ensure that: hour is between 0 and 23, minute is between 0 and 59, second is between 0 and 59. If any of the values are out of range, an error message is printed. Otherwise, the function retrieves the current RTC time into a rtc_time_t structure using get_rtc(&now). It then updates the hour, minute, and second fields with the new values. Finally, it calls set_rtc with the updated time values (while keeping the day, month, and year unchanged) and prints a success message indicating that the time has been updated successfully. As of R4, it was adjusted to also accept times formatted as “HH:MM:SS” to be more consistent with the alarm command. If a time is provided in this format, it will tokenize it so the time is internally stored the same as it was prior to R4 before functioning the exact same.		

void cmd_date(char** toks, int token_count);	Void function, dispatcher for date related commands	This function manages date commands. First, it checks if a second argument is provided. If not, it prints an error message indicating that the date command requires a second argument and suggests using "help date" for proper usage. If the second token is "get", the function calls cmd_date_get() to retrieve and display the current date. If the token is "set", it calls cmd_date_set(toks, token_count) to update the date based on additional tokens. For any other value of the second token, it prints an error message indicating an invalid argument and provides guidance to use "help date".	none	**char** toks**: Array of string representing the tokens from command line input. Expected tokens ‘ date, get or set, others’ **int token_count**: An integer indicating the total number of tokens provide in the ‘toks’ array
void cmd_date_get();	Void function, retrieves and displays current time	This function gets the current date. First, it declares a variable now of type rtc_time_t to store the current date and time, then calls get_rtc(&now) to retrieve the current RTC values. After obtaining the current date, it prints the month, day, and year using printf with the format specifier %2d/%2d/%2d,	none	none

		ensuring a consistent two-digit display for each component.		
void cmd_date_set(char** toks, int token_count);	Void function, sets the current system date based on user input	This function sets the current date by extracting the month, day, and year from the provided tokens. First, it checks that there are at least 5 tokens; if fewer tokens are provided, it prints an error message indicating the correct format ("date as MM DD YY"). If the required tokens are present, it converts the strings for the month, day, and year into integers. Next, it performs basic validation: it ensures that the month is between 1 and 12 and that the day is within the valid range for that month (using the month_length function with the provided month and year). If any of these checks fail, it prints an "Invalid date values" error. Otherwise, it retrieves the current RTC values into a structure using get_rtc(&now), updates the month, day, and year fields of that structure with the new values, and then calls set_rtc with the updated date while retaining the current time values (hour, minute, second). Finally, it prints a	none	**char** toks:** Array of string representing tokens from the command line. Expected tokens ‘date, set, month value, day value, year value’ **int token_count**: Total number of tokens provided from the ‘toks” array

		success message indicating that the date was updated successfully. As of R4, it was adjusted to also accept dates formatted as “MM/DD/YY” to be more consistent with the alarm command. If a date is provided in this format, it will tokenize it so the date is internally stored the same as it was prior to R4 before functioning the exact same.		

struct pcb {
  char name[9];
  unsigned int class_type : 1;
  unsigned int suspended : 1;
  unsigned int state : 2; // ready, running, blocked
  unsigned int priority : 4; // 0-9
  struct pcb *next_ptr;
  struct pcb *prev_ptr;
  unsigned char *stack_ptr;
  unsigned char stack[1024];
};
	PCB Structure, represents a PCB	This structure is designed to encapsulate all relevant information about a process in a compact and efficient manner.
		
		First, it stores a fixed-length name (up to 8 characters plus a NUL terminator) that uniquely identifies the process. It uses bit fields for the class_type, suspended, state, and priority members to minimize memory usage while representing small-range values efficiently.
		The state field indicates whether a process is ready, running, or blocked, while the priority field (with a range of 0 to 9) assists in scheduling
			N/A	**char name[9]**: Holds the processes name, space for 8 characters and a NULL terminator.
				unsigned int class_type:1 :A 1-bit flag representing the process class (user or system)
				unsigned int suspend : 1 : A 1-bit flag representing whether the process is suspended
				unsigned int state : 2 : A 2-bit field representing the process state (ready, running, or blocked)
				unsigned int priority : 4 : A 4-bit field that specifies the processes priority level, range from 0 to 9
				struct pcb *next_ptr : Pointer to the next PCB in the doubly linked list
				struct pcb *prev_ptr: pointer to the previous PCB in the doubly linked list.
				unsigned char *stack_ptr: Pointer to the current position within the processes stack.
				unsigned char stack[1024]: An array reserved for the processes stack, provides a

		decisions. The next_ptr and prev_ptr fields are pointers used to create a doubly-linked list of PCBs, facilitating easy traversal and management of process queues. Additionally, the structure includes a dedicated stack area (stack) along with a stack_ptr to manage the process's execution context. Although the structure itself does not implement an algorithm, its design is critical for algorithms in the operating system that handle process scheduling, state transitions, and memory management.		fixed size memory area for process execution context.
typedef struct pcb pcb_t;	pcb_t typedef, allows for being able to just call pcb_t to refer to struct pcb	makes code readability better	N/A	N/A
extern pcb_t* pcb_ptrs[3];	External array of two pointers to pcb_t structures	to keep track of a limited number of active PCBs for quick access across different modules of the project.	N/A	Array size: contains two elements element type: each is a pointer to a pcb_t structure.

pcb_t* pcb_allocate(void);	pcb_t function, returns a pointer to a pcb_t structure	This function is designed to allocate memory for a new PCB and initialize its stack pointer. First, it calls sys_alloc_mem(sizeof (pcb_t)) to dynamically allocate the memory required for a PCB. If the allocation fails, the function returns NULL to indicate the error. If the allocation is successful, the function initializes the PCB's stack pointer by setting it to point near the top of the allocated stack area. Specifically, the pointer is set to pcb_ptr->stack + sizeof(pcb_ptr->stack) - 2, ensuring that the stack is correctly positioned for future use. Finally, the function returns the pointer to the newly allocated and initialized PCB.	Type: pcb_t* Returns a pointer to the allocated and initialized PCB	none
int pcb_free(pcb_t* pcb_ptr);	Integer function, used to free memory previously allocated for a PCB	This function releases the memory previously allocated for a PCB. First, it receives a pointer to the PCB (pcb_ptr). It then calls the system-level memory free function sys_free_mem, passing pcb_ptr as the argument. The function returns the result of	Type: Integer Returns a zero if it was a success and en error if something went wrong	**pcb_t* pcb_prt**: A pointer to the PCB that is to be freed

		sys_free_mem, indicating whether the memory was successfully freed or if an error occurred.		
pcb_t* pcb_setup(const char *name, int class_type, int priority);	pcb_t structure, returns a pointer to a pcb_t structure	This function sets up a new PCB with the specified attributes. First, it checks if the provided process name exceeds 8 characters; if it does, the function returns NULL to indicate an error, since the PCB structure only supports names up to 8 characters. Next, it allocates memory for the PCB by calling pcb_allocate(). If memory allocation fails, it returns NULL. Once a valid PCB pointer is obtained, the function copies the process name into the PCB's name field using memcpy, sets the class_type and priority fields based on the provided parameters (casting them to unsigned integers), and initializes the state field to PCB_READY, indicating that the process is ready to run. Finally, it returns the pointer to the fully initialized PCB.	Type: pcb_t* Returns a pointer to the initialized PCB if successful, returns 'NULL' if it fails	**const char *name**: Name of the process **int class_type:** indicates the class of process (user or system) **int priority**: Number representing the priority for the process

pcb_t* pcb_find(const char *name);	pcb_t structure, returns a pointer to a pcb_t structure	This function finds a PCB by its name. First, it calculates the number of elements in the pcb_ptrs array by dividing the total size of the array by the size of a pointer to pcb_t. It then iterates over each element in this array. For each non-NULL element, it traverses the linked list of PCBs using the next_ptr field. During the traversal, it compares the name field of each PCB with the provided name using strcmp(). If a match is found (i.e., strcmp returns 0), the function immediately returns the pointer to the matching PCB. If no matching PCB is found after searching all lists, the function returns NULL.	Type: pcb_t* Returns a pointer to the PCB if one is found with a matching name, returns NULL if not found	**const char *name**:String representing the name of the PCB to search for
void pcb_insert(pcb_t* pcb_ptr);	Void function, takes a pouter to a pcb_t structure and inserts the PCB into the appropriate process queue	This function inserts a PCB into the appropriate process queue based on its current state. First, it checks the state field of the provided PCB. If the PCB is in the PCB_READY state, the function calls pcb_insert_queue_prioritized with the PCB pointer and the address of the ready queue pointer (PCB_READY_PTR),	none	**pcb_t* pcb_ptr**: A pointer the the PCB that is to be inserted into the scheduling queue

		ensuring that the PCB is inserted in the correct position based on its priority. Alternatively, if the PCB is in the PCB_BLOCKED state, it calls pcb_insert_queue with the PCB pointer and the address of the blocked queue pointer (PCB_BLOCKED_PTR), adding the PCB to the blocked queue. This conditional insertion helps maintain the correct scheduling order and process management.		
void pcb_insert_queue_prioritized(pcb_t* pcb_ptr, pcb_t** pcb_queue_ptr);	Void function, inserts a PCB into a prioritized process queue	This function inserts a given PCB into a prioritized queue so that the PCBs remain sorted by priority. First, it checks if the queue is empty. If the queue is empty, it simply assigns the PCB to the head of the queue and returns. If the queue is not empty, it compares the priority of the new PCB with the priority of the PCB currently at the head. If the new PCB has a lower numerical priority (higher priority) than the head, it is inserted at the beginning by adjusting the pointers accordingly. If the new PCB does not have the highest priority, the function	none	**pcb_t* pcb_ptr:** A pointer to the PCB that is going to be inserted into the prioritized queue **pcb_t** pcb_queue_ptr**: A pointer to the pointer representing the head of the prioritized queue

		iterates through the linked list until it finds the correct insertion point—specifically, the position just before a PCB whose priority is greater than the new PCB's priority. At this point, it calls pcb_insert_spot to insert the PCB at the identified location. If the end of the list is reached without finding a PCB with a higher priority, the new PCB is inserted at the end of the queue.		
void pcb_insert_queue(pcb_t* pcb_ptr, pcb_t** pcb_queue_ptr);	Void function, inserts a PCB into a non-prioritized process queue	This function inserts a PCB into a non-prioritized queue by adding it at the end. First, it checks whether the queue is empty by testing if the head pointer (*pcb_queue_ptr) is NULL. If the queue is empty, the function sets the head to the new PCB and returns immediately. If the queue is not empty, the function traverses the linked list by iterating through the next_ptr pointers until it reaches the last PCB (i.e., a PCB whose next_ptr is NULL). Once the last element is reached, the function calls pcb_insert_spot, passing the new PCB	none	**pcb_t* pcb_ptr**: A pointer to the PCB that is going to be inserted into the queue **pcb_t** pcb_queue_ptr**: A pointer to the pointer representing the head of the queue

		and the last element as arguments, to properly insert the new PCB into the list. This ensures that the PCB is appended at the end of the queue.		
void pcb_insert_spot(pcb_t* pcb_ptr, pcb_t* pcb_prev_ptr);	Void function, inserts a PCB into a specific spot in a doubly linked list.	This function inserts a new PCB into an existing doubly-linked list immediately after a specified PCB. First, it sets the new PCB's next_ptr to point to what was previously the next element after pcb_prev_ptr. Then, it updates the new PCB's prev_ptr to point to pcb_prev_ptr. Next, it makes pcb_prev_ptr point to the new PCB by updating its next_ptr. Finally, if there is a PCB following the newly inserted one, the function updates that PCB's prev_ptr to point back to the new PCB. This careful manipulation of pointers ensures that the doubly-linked list remains correctly connected after the insertion.	none	**pcb_t* pcb_ptr**: A pointer to the new PCB that is to be inserted into the list. **pcb_t* pcb_prev_ptr**: A pointer to the PCB after which the new PCB is to be inserted.
int pcb_remove(pcb_t* pcb_ptr);	Integer function, used to remove the specified pointer both from the global array of pcb pointers and the doubly linked list of pointers	This function removes a given PCB from the system. First, it iterates through the global PCB pointer array (pcb_ptrs) to check if	Type: int Returns a 0 if removal was a success.	**pcb_t* pcb_ptr**: A pointer to PCB to be removed

		the PCB to be removed is directly referenced there; if a match is found, that element of the array is updated to point to the next PCB in the list. Next, it checks if the PCB has a previous node in its doubly-linked list. If so, it updates the previous node's next_ptr to bypass the PCB being removed. Similarly, if the PCB has a next node, the function updates that node's prev_ptr to point to the PCB's previous node. This careful pointer update ensures that the doubly-linked list remains intact after the removal. Finally, the function returns 0 to indicate successful removal.		
void print(const char* str);	Void function, outputs a given string	This function prints NUL-terminated strings to the console.. First, it calculates the length of the input string using strlen(str). Then, it calls the system request function sys_req with the parameters: (WRITE, COM1, str, strlen(str))	none	**const char* str:** A pointer to a NUL-terminated string that is to be printed.

void printf(const char* str, ...);	Variadic Void function, outputs a formatted string.	This function prints a NUL-terminated format string to the console. First, it parses a format string and processes variable arguments. It prints literal text until a % is encountered, then handles the conversion specifier—printing a literal %, a string (%s), or an integer (%d or %i) with appropriate width—before continuing. Finally, it prints any remaining text and cleans up the argument list.	none	**const char* str**: A NUL-terminated format string containing literal text and format specifiers. **… Variadic arguments**: a variable number of additional arguments corresponding to conversion specifiers present in the format string.
int input(char** toks, int token_size, char* buf, int buffer_size);	Integer function, returns a int representing the number of tokens parsed from user input.	This function prompts for input from the console, then tokenizes said input for processing. First, this function writes a prompt ("> ") to COM1 and reads input into a buffer. It converts the input to lowercase and tokenizes it using whitespace as the delimiter, storing each token in the provided array. The function returns the number of tokens found, or -1 if the input read fails or is empty.	Type:int Returns the number of tokens parsed or -1 if it fails or is empty	**char** toks**: An array where the pointers to tokens will be stored **int token_size**: The maximum number of token that can be stored in 'toks' **char* buf**: Buffer used to store the raw input string **int buffer_size**: The size of input buffer 'buf'

void create_pcb(const char* name, int class_type, int priority);	Void function, creates and inserts a new PCB	This function calls pcb_setup with the given parameters to allocate and initialize a new PCB. If pcb_setup returns a valid PCB pointer (i.e., not NULL), it then inserts the PCB into the system's process queue using pcb_insert.	none	**const char* name**: The name of the new PCB **int class_type**:Integer representing the class type of PCB (user or system) **int priority**: integer specifying the priority of PCB
void delete_pcb(const char* name);	Void function, will delete the named PCB	This function deletes a PCB First it locates a PCB using its name. It calls pcb_find to search for the PCB; if the PCB is found, it proceeds to remove it from any queues by calling pcb_remove and then frees its allocated memory using pcb_free.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB that is to be deleted
void block_pcb(const char* name); **This function is removed as of R6**	Void function, will change a PCB's state to blocked.	This function blocks a PCB. First, it locates a PCB using pcb_find. If the PCB is found, it removes the PCB from its current queue with pcb_remove, sets its state to PCB_BLOCKED, and then reinserts it into the appropriate queue using pcb_insert.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB that is to be blocked

void unblock_pcb(const char* name); **This function is removed as of R6**	Void function, will unblock the named PCB	This function unblocks a PCB. First, it locates a PCB using pcb_find. If found, it removes the PCB from its current queue via pcb_remove, sets its state to PCB_READY, and then reinserts it into the appropriate queue with pcb_insert.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB that is to be unblocked
void suspend_pcb(const char* name);	Void function, that will suspend the named PCB	This function suspends a PCB First, it locates a PCB using pcb_find. If the PCB is found, it is removed from its current scheduling queue via pcb_remove. The function then sets the PCB's suspended flag to PCB_SUSPENDED to indicate that the process is suspended. Finally, it reinserts the PCB into the appropriate queue with pcb_insert, ensuring that the PCB's new suspended status is recognized by the system.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB that is to be suspended
void resume_pcb(const char* name);	Void function, that will resume the named PCB	This function resumes a PCB. First, it locates a PCB using pcb_find. If the PCB is found, it is removed from its current scheduling queue using pcb_remove. The function then updates	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB that is to be resumed

		the PCB's suspended flag by setting it to PCB_NOT_SUSPENDED to indicate that the process is no longer suspended. Finally, the PCB is reinserted into the appropriate queue using pcb_insert, ensuring that the process is now available for scheduling.		
void set_pcb_priority(const char* name, int priority);	Void function, that will change the priority of the named PCB to the given priority integer	This function updates the priority of a PCB. First, it locates a PCB using pcb_find. If the PCB is found, it removes the PCB from its current scheduling queue with pcb_remove, updates the priority field with the new value, and reinserts the PCB into the queue using pcb_insert to ensure that the new priority is reflected in the process scheduling order.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB which user would like to change priority of **int priority**: The new priority value to be assigned to the PCB
void show_pcb(const char* name);	Void function, will display information about the named PCB	This function prints a PCB to the console given its name. First, it locates a PCB using pcb_find. If the PCB is found, it calls show_pcb_t to display the PCB's details. If no matching PCB is found, the function performs no further action.	none	**const char* name** : A NUL-terminated string that specifies the name of the PCB to be displayed

void show_pcb_t(pcb_t* pcb_ptr);	Void function,	This function displays a PCB to the console given a pointer to it. First, it prints the PCB's name followed by a tab. It then determines the printable form of the class type (either "SYSTEM" or "USER") and prints it, followed by another tab. Next, it converts the PCB's state to a string ("READY", "BLOCKED", or "RUNNING") and prints it with a tab. The function then does the same for the suspended status, printing "SUSPENDED" or "NOT SUSPENDED" followed by a tab. Finally, it converts the priority value to a string and prints it, ending with a newline.	none	**pcb_t* pcb_ptr**: A pointer to the PCB whose details are to be displayed
void show_ready(void);	Void function, will display all PCB's in the ready queue	This function displays all PCBs that are in the ready state. This function calls show_queue with the head of the ready queue and the row name "READY :"	none	none
void show_blocked(void);	Void function, will display all PCB's that are currently blocked	This function displays all PCBs that are in the blocked state. This function calls show_queue with the head of the blocked queue and the row name "BLOCKED:"	none	none

void show_all(void);	Void function that will display all PCB's	This function prints the details of every PCB stored in the global array. This function calls show_running, show_ready, and show_blocked, thus showing all PCBs.	none	none
R3 ADDITIONS				
void cmd_pcb_yield	Void function, Process handling **This function has been removed as of R4.**	This function yields the CPU to allow other processes to run. It 'yields' the CPU by calling the sys_req(IDLE) function. This causes the command handler to voluntarily give up the CPU for usage elsewhere. It then prints a message to show it is yielding.	none	none
void cmd_pcb_loadR3	Void function, Process handling	This function loads the test processes outside of the comhand for testing purposes. This function first assigns names to the test processes, then iterates over the 5 using a for loop. Inside, it allocates and initializes the processes and assigns their context. This is done using the context struct, covered below, which is assigned the appropriate initialization flags.	none	none

char* strcpy(char* dest, const char* src)	Process Handling, Copies NULL terminated string to another mem location	This function copies a NULL terminated string to another location in memory This function uses a while loop to iterate over the source string, filling the destination string until the NUL terminator is found.	**char*** dest; pointer to new string location	**char* dest:** location to copy into **Const char* src:** location to copy from

struct context {
int32_t ds;
int32_t es;
int32_t fs;
int32_t gs;
int32_t ss;
int32_t eax;
int32_t ebx;
int32_t ecx;
int32_t edx;
int32_t esi;
int32_t edi;
int32_t ebp;
int32_t eip;
int32_t cs;
int32_t eflags;
};
typedef struct context context_t;
	Structure, Process Handling	This structure stores the flags associated with the context of a given process.	This struct can be called on with the variable context_t	The associated elements of this struct use int32_t. They are used as flags with their associated usage in terms of the context of a process.

context_t *sys_call(context_t * current_context_ptr)	Function, Process handling	This function performs the actual context switch from one process to another. **Must be called from sys_call_isr.s or else it won't work.** This function uses the eax flag within the current context to determine in what way to handle the current process. I.e. if the op code is idle then it will	This function returns the context of the next process set to run.	**context_t * current_context_ptr:** The context of the last running process, which the eax flag is used to determine the way to switch context.

		attempt to insert, if it is exit then it will attempt to free. Once this is complete it will create the context of the next process, deriving the stack pointer from the currently running process.		
R4 ADDITIONS				
char *strcat(char *dest, const char *src)	Char* Function, String Processing	Function that performs a simple string concatenation, copying src to the end of dest. The function assumes there's enough memory allocated to dest to do this, and has no error checking of its own. First, the end of the dest string is found by iterating until a NUL-terminator is found. Next, each character in src is iterated through, copying each to the end of dest.	Returns the argument dest.	**char *dest:** string concatenated onto. **const char *src:** string concatenated onto dest.

typedef struct {
  int hour;
  int minute;
  int second;
  char message[80];
  char pcb_name[12];
  int active;
} alarm_data_t
	Structure, Alarms	This structure holds all data related to a given clock outside of the clock process itself.
		
		The clock process, instead of holding all this information itself as variables, instead has a single variable that is a pointer to an alarm data struct inside a global array of all alarm data structs.
			N/A	**int hour:** hour of the day the alarm goes off, military time
				int minute: minute of the hour the alarm goes off
				int second: second of the minute the alarm goes off
				char message[80]: message the alarm displays when going off
				char pcb_name[12]: name of the pcb corresponding to the alarm. Also displayed when the alarm goes off.
				int active: boolean indicator of if a spot in the alarm data

				array is taken up by an active alarm. 1 if in use, 0 otherwise
static alarm_data_t g_alarms[MAX_ALARMS]	Alarm_data_t Array Variable, Alarms	Global array of alarm data. An alarm process, instead of holding alarm data itself as variables, instead holds a single variable that is a pointer to an entry in this array.	N/A	N/A
static int g_alarm_count = 0	Int Variable, Alarms	Count of entries in the global alarm array that are currently in use. Used to confirm there's enough space to create a new alarm.	N/A	N/A
void cmd_alarm(char **toks, int token_count)	Void Function, Alarms	Function that creates a new alarm by interpreting supplied command line arguments. The function checks for sufficient arguments, check for enough space to make another alarm, parses the time argument (given as “HH:MM:SS”) for timestamps, validates timestamps, creates and stores an alarm struct, and finally creates and stores a PCB struct including a context struct for the new process.	none	**char** toks:** Array of string representing tokens from the command line. toks[1] is time for the alarm to go off in the form of “HH:MM:SS”, and toks[2] is the message to be displayed by the alarm. **int token_count**: Total number of tokens provided from the “toks” array

void alarm_proc(void)	Void Function, Alarms	Main function body of alarm processes. First, the process gets the pointer to the alarm data struct from register value EBX. Next, it checks the current time to see if its time to go off. If so, the process displays its message and then exits. Otherwise, it idles before repeating.	none	none
void show_queue(pcb_t* queue_ptr, const char* row_name);	Void function, will display all PCB's of a queue	This function displays all PCBs that are in a queue. It initializes a pointer to the head of the supplied queue and enters a loop. For each PCB in the queue, it calls show_pcb_t to print its details, then advances the pointer to the next PCB until the end of the list is reached.	none	**pcb_t* queue_ptr:** pointer to PCB at head of queue to print (or, if queue is empty, NULL) **const char* row_name:** string to prepend to first PCB indicating the kind of queue being printed.
void show_running(void)	Void function, will display all PCB's of type running	This function displays all PCBs that are in the running queue. This function calls show_queue with the head of the running queue and the row name "RUNNING:"	none	none

void cmd_pcb_ showrunning()	Void function, displays a list of all PCB's in a running state	This function displays the list of running PCBs. First, it prints a table header using the PCB_TABLE_HEAD constant to label the output columns. Next, it calls show_running() to output the details of each PCB that is in the blocked state.	none	none
R5 ADDITIONS				

struct mcb {
  struct mcb* prev_ptr;
  struct mcb* next_ptr;
  unsigned int allocated : 1;
  char* start_ptr;
  size_t size;
 	
};
typedef struct mcb mcb_t;
	Structure, MCBs	This structure contains all the relevant information for managing the MCBs.
		
		This struct allows MCBs to be created and be indexed using the elements of the struct.
			none	**struct mcb* prev_ptr:** Pointer to the previous memory block.
				struct mcb* next_ptr; Pointer to the next memory block
				unsigned int allocated; 1 if block in use, 0 if free
				char* start_ptr; Pointer to the start of memory block
				size_t size; Size of memory block in bytes

void initialize_heap(size_t size);	Void, MCBs	Initializes the heap of a given size. Allocates a memory region using kmalloc() and establishes the first MCB with struct mcb_t. Also sets indexing pointers to NULL, marks the memory as free, and sets the start pointer.	none	**Size_t size;** Total size in bytes of memory to initialize.

void* allocate_memory(size_t size);	Void, MCBs	Implements the first fit memory allocation - searches for free memory block of fitting size. Starts at the heap_ptr and iterates over the mcb_t blocks, until one is found. Includes an if statement to split the given block if there is enough to match the minimum value for a free block. The block is marked as allocated.	Returns a pointer to the start of the allocated memory, or NULL if none is found.	**Size_t size;** Number of bytes to be allocated
int free_memory(void* memory_ptr);	MCBs	Frees a block of memory. Calculates the associated block by subtracting the MCB size from the given memory pointer. Iterates over the linked list from the heap_ptr, validating that both that it exists and is allocated. Additional if statement ensures the block is not tied to an active process. Finally, the block is set to free, and nearby blocks are assessed to see if they can be merged with mcb_merge().	Returns an int, 0 if success, -1 if unknown pointer, -2 if block is already free, -3 if block is in use by a process	**void* memory_ptr;** The pointer to a given chunk of memory to be freed, as given by allocate_memory

mcb_t* mcb_split(mcb_t* mcb_ptr, size_t size);	Internal helper function; MCBs	Splits a given memory block into an allocated and free block, if the size of the block is large enough to warrant splitting. The new MCB address is calculated after the new allocation, and its pointers are set. The original mcb_ptr is updated to reflect the smaller size. The blocks are then linked in the linked list.	Returns a pointer to the remaining free portion after the split.	**mcb_t* mcb_ptr;** The pointer to the MCB being split. **Size_t size;** The desired size of the allocated portion
mcb_t* mcb_merge(mcb_t* mcb_ptr);	Internal helper function; MCBs	Used to combine given adjacent free blocks of memory. Adds the next free portion of memory (sizeof(mcb_t)) and size to mcb_ptr->size, so as to have one continuous space. Updates the pointers to unlink the merged MCB.	Returns the merged mcb_ptr	**mcb_t* mcb_ptr;** Pointer to a free block of memory
pcb_t* pcb_find_addr(void* addr);	Internal helper; MCBs	Checks to see if a given address corresponds to any active PCBs. Prevents deallocation of memory in use. Iterates over all the different pcb types, and for each non-NULL, it compares addr with each pcb_t*.	Returns NULL if no match, pointer to the matching process (pcb_t*) if found.	**Addr;** The desired address of memory to be searched for among the pcbs.

void cmd_mcb (char** toks, int token_count);	Void, Command handler	This function interprets commands and omits those missing arguments at a high level. Interprets the toks of a given input and calls the appropriate command, otherwise giving an error message.	None	**char** toks;** the array of entered string tokens **Int token_count;** The total number of tokens passed
void cmd_mcb_free(char** toks, int token_count);	Void, subcommand	This command handles the user side of freeing memory. Validates the token contains valid hex string, and converts using hex_to_uintptr(), which is then passed to free_memory(). Return depends on code sent by function.	None	**char** toks;** the array of entered string tokens **Int token_count;** The total number of tokens passed
void cmd_mcb_allocate(char** toks, int token_count);	Void, subcommand	This command handles the user side of allocating memory. Validates the third token and the provided string size, and then converts it using atoi(). Calls allocated memory using the converted value. Prints error message or the address of the newly allocated memory.	None	**char** toks;** the array of entered string tokens **Int token_count;** The total number of tokens passed

void cmd_mcb_show(int allocated);	Void, subcommand	Starts at the heap_ptr to iterate through the MCBs. Contains a filter to either show allocated, free, or all based on the token given. Prints a table header and the data of the desired MCBs. Uses a standard current_pointer method to iterate through the list.	None	**Int allocated;** 1 if show allocated, 0 if show free, -1 if all (passed by cmd_mcb)
uintptr_t hex_to_uintptr(const char *hex_str);	Uintptr_t, internal helper	Converts a string-given hex number into an int. Checks for the typical 0x prefix of hex and removes it. Iterates over the string and converts given hex digits to their int. Does so by accumulating the result, by multiplying by 16 and adding the calculated digit.	Returns the uintptr_t of the hex string	**Const char* hex_str;** Null terminated string representing a hex number.
R6 ADDITIONS				
int serial_open(device dev, int speed);	Device initialization / Serial communication	Opens and initializes a serial device for communication. Performs validation checks on device ID and baud rate, initializes its Device Control Block (DCB), installs interrupt service routines (ISRs), configures line control settings, enables necessary hardware interrupts, and marks the device as initialized.	Returns int of, 0 on success. -101 if device is invalid. -102 if baud rate is unsupported.	**device dev;** Identifier for the serial device to initialize. **int speed;** Desired baud rate (must be in valid_bauds)**.**

int serial_close(device dev);	Device deinitialization / Serial communication	Closes a previously initialized serial device. Validates that the device was initialized, sets its DCB state to closed, disables its associated IRQ in the PIC mask, disables hardware interrupts, and clears the initialization flag.	Returns int of, 0 on success. -201 if device is invalid	**device dev:** device to close.
int serial_read(device dev, char *buf, size_t len);	I/O operation / Serial communication	Initiates a read operation from the serial device's ring buffer into the user's buffer via the associated IOCB. Validates device state, checks if data is available, and reads characters until the buffer is full or an end-of-line character is found. Updates process state and device flags when read completes.	Returns int of characters read into the buffer or 0 if more data is needed. -301 if device is invalid or not open. -304 if device is busy.	**device dev:** serial device to read from.
int serial_write(device dev, char *buf, size_t len);	I/O operation / Serial communication	Begins writing data from the process buffer to the serial device. Validates device status, sets DCB to writing mode, transmits the first character to the device's transmit holding register (THR), and enables write interrupts to handle the rest of the data asynchronously.	Returns int; 0 on successful initiation of write. -401 if device is not initialized or not open. -404 if device is busy.	**device dev:** serial device to write to

void serial_interrupt(void);	ISR handler / Serial communication	Handles incoming serial hardware interrupts. Identifies which UART raised the IRQ, determining the interrupt type via the IIR, and dispatching to the appropriate handler (serial_input_interrupt or serial_output_interrupt). It processes only one active device per call and ends by sending an EOI to the PIC.	N/A	**N/A**
void serial_input_interrupt(dcb_t *dcb_ptr);	ISR helper	Handles a received character from the serial port during an input interrupt. If a user read is active, the character is placed into the IOCB buffer; if a newline or buffer limit is reached, the read is completed and dequeued. If no read is active, the character is placed into the device's ring buffer if there is room; otherwise, it is dropped.	N/A	**dcb_t *dcb_ptr:** Pointer to the device's control block that maintains the current read state
void serial_output_interrupt(dcb_t *dcb_ptr);	ISR helper	Handles output interrupts triggered when the transmitter is ready. If a write is in progress, it sends the next byte from the IOCB buffer. When all characters are sent, it disables transmit	N/A	**dcb_t *dcb_ptr:** Pointer to the device's control block that maintains the current read state

		interrupts and signals completion by calling dcb_dequeue().		
static unsigned short dev_from_dno(int dno)	Static unsigned short, helper function	This function takes a dno int, representing the port index, and returns the corresponding COM. It works by using a switch, which has cases for each of the 4 DNOs, and returns the appropriate COM port base address.	Returns a static unsigned short, the given COM port address	**Int dno;** the int value of the port index, should be a value between 0-3.

typedef struct {
  device dev;
  unsigned int open : 1;
  unsigned int event_flag : 1;
  unsigned int status : 2;
 	
  // intermediate ring buffer
  char ring_buffer[RING_BUFFER_SIZE];
  unsigned int ring_in : 8;
  unsigned int ring_out : 8;
 	
  struct iocb* iocb_ptr;
 	
} dcb_t;
	Structure, DCB	Device control block used to manage the state of each serial port. Tracks whether the device is open, its status, an event flag, intermediate ring buffer state, and a pointer to the current IOCB.
		
		Used for all serial functions and interrupt handlers to coordinate access and operation.
			N/A	**device dev:** serial device
				unsigned int open: 1 if open, 0 if closed.
				unsigned int event_flag: Set to 1 when an operation completes.
				unsigned int status: 2-bit field (IDLE, READING, WRITING)
				char ring_buffer[]: Internal character queue.
				unsigned int ring_in, ring_out: Ring buffer head/tail indices.
				struct iocb* iocb_ptr: Pointer to current I/O operation.

struct iocb {
pcb_t* pcb_ptr;
dcb_t* dcb_ptr;
unsigned int operation : 2;
unsigned int event_flag : 1;
 	
char* buffer;
size_t buffer_size;
size_t buffer_pos;
 	
struct iocb* next_ptr;
struct iocb* prev_ptr;
};
typedef struct iocb iocb_t;
	Structure, IOCB	Represents a pending I/O operation, either read or write, and is linked to both the initiating process (pcb_t) and the target device (dcb_t).	N/A	**pcb_t* pcb_ptr:** Pointer to requesting process
				dcb_t* dcb_ptr: Associated device control block handling the I/O.
				unsigned int operation: Operation type, read or write
				unsigned int event_flag: Set to 1 when I/O is complete.
				char* buffer: buffer for the data to be stored
				size_t buffer_size: Maximum size of buffer.
				size_t buffer_pos: Current write/read position in buffer.
				iocb_t* next_ptr, prev_ptr: Pointers for queueing IOCBs.

char dcb_ring_pop(dcb_t* dcb_ptr);	Internal Ring, DCB	Removes and returns a character from the ring buffer of the given DCB. Always advances the ring_out pointer to maintain state consistency, works by using pointer logic along with the modulo operator.	Returns char; at the current ring_out index,even if invalid if the buffer was empty.	**dcb_t* dcb_ptr:** Pointer to the device control block whose ring buffer will be read from
int dcb_ring_push(dcb_t* dcb_ptr, char val);	Internal Ring, DCB	Attempts to insert a character into the ring buffer of a DCB. Stores the character at the current ring_in index and advances the pointer. If the buffer is full, the character is discarded. Again uses pointer logic with the modulo operator.	Returns int; 0 on success, -1 if the buffer is full and the character is discarded.	**dcb_t* dcb_ptr:** Pointer to the device control block. **char val:** Character to be inserted into the ring buffer.

void dcb_enqueue(dcb_t* dcb_ptr, iocb_t* iocb_ptr);	Queuing, IOCB	Adds an iocb_t to the end of the queue of pending I/O operations for the specified DCB. If the queue is empty, the IOCB becomes the head of the queue; otherwise, it is appended to the end.	N/A	**dcb_t* dcb_ptr:** Pointer to the device control block. **iocb_t* iocb_ptr:** Pointer to the IOCB being enqueued.
void dcb_dequeue(dcb_t* dcb_ptr);	Queuing, IOCB	Removes the head IOCB from the DCB I/O operation queue, signaling that the operation is complete. It updates the associated PCB and performs necessary cleanup. If there are more operations in the queue, it starts the next one.	N/A	**dcb_t* dcb_ptr:** Pointer to the device control block.
int dcb_lock(device dev);	Locking, IOCB	Attempts to lock the device associated with the given dev identifier for exclusive use by the current I/O operation. The function checks if the device is idle and if no other processes are currently locking it. If successful, it increases the locking level (via incrementing) or assigns the locking process. (via pointer logic in the linked lists)	Returns int; 0 if the lock is successfully acquired. -501 if the device is not initialized/not open. -504 if the device is busy.	**device dev:** serial device to lock

int dcb_unlock(device dev);	Locking, IOCB	Attempts to unlock the device, freeing it for other processes. The function checks if the device is idle and if no other operations are ongoing before decrementing the lock level or clearing the locking process.	Returns int; 0 if the unlock operation is successful. -601 if the device is not initialized/not open. -604 if the device is busy.	**device dev:** serial device to unlock
char* get_private_interrupt_stack(void);	Helper function	Used to aid in interrupt handling Returns a pointer to the end of the private interrupt stack (just before the last two bytes).	N/A	**N/A,** (Utilizes the array sharing the same name)