File: Project1/testinput

```
help
display #address
n
write #address 0xFF
n
invert #address
n
prng #address
n
validate #address
n
allocate 0
allocate 10
allocate 10
display #address 10
write #address 0xFF00
invert #address
display #address
prng #address 10
validate #address 10
write #address 0xFFFF
validate #address 10
prng #address 10 200
display #address 10
invert #address 10
validate #address 10 200
invert #address 10
validate #address 10 200
free
free
exit
```

File: Project1/testing_script

```
grep -o '^[^//]*' commented_testinput > testinput
./P1_Memtester < testinput
```

File: Project1/makefile

```
# Heavily Influenced by below for automatically indexing/adding .c and .h files
# https://hiltmon.com/blog/2013/07/03/a-simple-c-plus-plus-project-structure/
#

CC := gcc
CFLAGS := -Wall
SRCPATH := src
BUILDPATH := build
INCPATH := include
TARGET := P1_Memtester

SRCEXT := c
SRC := $(shell find $(SRCPATH) -type f -name *.$(SRCEXT))
OBJ := $(patsubst $(SRCPATH)/%, $(BUILDPATH)/%, $(SRC:.$(SRCEXT)=.o))


$(TARGET):$(OBJ)
$(CC) $^ -o $@ $(LIB)

$(BUILDPATH)/%.o:$(SRCPATH)/%.$(SRCEXT)
mkdir -p build
$(CC) $(CFLAGS) -I $(INCPATH) -c -o $@ $<

.PHONY:clean

clean:
$(RM) -r $(BUILDPATH) $(TARGET)

test: $(TARGET)
bash testing_script
```

File: Project1/src/commandtable.c

```
1  /*
2   * commandtable.c
3   * @brief arrays to hold commands and functions to create and interact with them
4   * @author D.Doty
5  */
6
7  // Note: not sure if I'm allowed to use string functions or if I have to write my own.
8  // Using lib for now, will change if necessary
9
10 /* Includes */
11
12 #include "commandtable.h"
13
14 /* Defines */
15
16
17 /* Global Variables */
18 char** command_human = NULL;
19 char** command_help = NULL;
20 command_proto* command_table = NULL;
21 uint8_t command_quantity = 0;
22
23 /* Private Function Prototypes */
24
25
26 /* Function Definition */
27 // Creates tables to hold the user command names, help messages, and function pointers
28 int8_t command_table_init(uint8_t num_of_commands)
29 {
30  // Don't allow initialization if it's already been done
```

```
31  if(command_quantity)
32  {
33   printf("ERROR:Command Table Double Inititialization\n");
34   return 1;
35  }
36
37  // Store size of table
38  command_quantity = num_of_commands;
39
40  // Allocate the array of strings for human readable commands
41  command_human = (char**)malloc(command_quantity * sizeof(char*));
42  if(command_human)
43  {
44   printf("ERROR:Command Table Initialization, Human Readable Array\n");
45   return 1;
46  }
47
48  // Allocate the array of strings for command help messages
49  command_help = (char**)malloc(command_quantity * sizeof(char*));
50  if(command_help)
51  {
52   printf("ERROR:Command Table Initialization, Help Messages\n");
53   return 1;
54  }
55
56  // Allocate the array of function pointers
57  command_table = (command_proto*)malloc(command_quantity * sizeof(command_proto*));
58  if(command_table)
59  {
60   printf("ERROR:Command Table Initialization, Function Pointer Array\n");
61   return 1;
62  }
63
64  return 0;
65 }
66
67 // Adds a command to the table created with command_table_init
68 int8_t add_command(char* human_name, char* help_msg, command_proto func_pointer)
69 {
70  static uint8_t command_table_index = 0;
71
72  // Catch too many functions added to table
73  if(command_table_index > (command_quantity-1))
74  {
75   printf("ERROR:More Commands Initialized Than Table Size\n");
76   return 1;
77  }
78
79  // Allocate and fill the Human Readable Command Name
80  command_human[command_table_index] = (char*)malloc(strlen(human_name + 1)); //plus one for null
81  strcpy(command_human[command_table_index], human_name);    //copy the fed name into the array
82
83  // Allocate and fill the Help Message array
84  command_help[command_table_index] = (char*)malloc(strlen(help_msg + 1)); //plus one for null
85  strcpy(command_help[command_table_index], help_msg);    //copy the help message in
86
87  // Fill the function pointer in
88  command_table[command_table_index] = func_pointer;    //copy the func pointer into the table
89
90  // Index
91  command_table_index++; //increment the table so it will fill the next slot next time
92
93  return 0;
94 }
```

File: Project1/src/display.c

```
1   /*
2    * display.c
3    * @brief function to display a memory location
4    * @author D.Doty
5   */
6
7   /* Includes */
8   #include "display.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t display(char* args)
21 {
22  // Parse arguments
23  uint64_t address = 0;
24  uint64_t word_qty = 1;
25  io_parse(args, 2, &address, &word_qty);
26
27  // Check if the provided memory address and word quantity are in allocated block
28  // Get user confirmation to proceed if they're out of range
29  if(valid_range(address, word_qty) != 0)
30  {
31   // User has indicated they don't want to proceed. Returns to main for a new command
32   return 1;
33  }
34
35  // Display
36  printf("Word#\tAddress\t\t\t\tContents Hex\tContents Decimal\n");
37
38  for(uint8_t i = 0; i < word_qty*sizeof(uint32_t); i += sizeof(uint32_t))
39  {
40   printf("%lu\t0x%016lX\t0x%08X\t%u\n",
41     i/sizeof(uint32_t),     // Block #
42     address + i,      // Address
43     *((uint32_t*)(address + i)),  // Address Contents Hex
44     *((uint32_t*)(address + i)));  // Address Contents Decimal
45  }
46  printf("\n");
47  return 0;
48 }
```

File: Project1/src/invert.c

```
1   /*
2    * invert.c
3    * @brief inverts a memory range
4    * @author D.Doty
5   */
```

```
6
7  /* Includes */
8  #include "invert.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t invert(char* args)
21 {
22  // Parse arguments
23  uint64_t address = 0;
24  uint64_t word_qty = 1; // Default. If no word qty supplied, assumed user just wants to invert the supplied address
25  io_parse(args, 2, &address, &word_qty);
26
27  // Check if the provided memory address and word quantity are in allocated block
28  // Get user confirmation to proceed if they're out of range
29  if(valid_range(address, word_qty) != 0)
30  {
31   // User has indicated they don't want to proceed. Returns to main for a new command
32   return 1;
33  }
34
35  // Invert
36  printf("Inverting...\n");
37  clock_t begin = clock();
38
39  // Loop through specified address range inverting each entry
40  for(uint64_t i = 0; i < word_qty*sizeof(uint32_t); i += sizeof(uint32_t))
41  {
42   *((uint32_t*)(address + i)) ^= 0xFFFFFFFF;
43  }
44
45  clock_t end = clock();
46  // Calculate execution time in uS to avoid floating point
47  uint64_t execution_time = (1000000*(uint64_t)(end - begin))/CLOCKS_PER_SEC;
48  printf("Execution time: %lu uS\n\n", execution_time);
49  return 0;
50 }
```

File: Project1/src/allocate.c

```
1  /*
2   * allocate.c
3   * @brief command to allocate memory
4   * @author D.Doty
5  */
6
7  /* Includes */
8  #include "allocate.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t allocate(char* args)
21 {
22  uint64_t qty_words = 0;
23  io_parse(args, 1, &qty_words);
24
25  // Check if # of words was left out or set to 0
26  if(qty_words == 0)
27  {
28   printf("Cannot allocate 0 words\n\n");
29   return 1;
30  }
31
32  // Check if a memory block has already been allocated
33  if(block_ptr != NULL)
34  {
35   printf("Memory has already been allocated. Please free before allocating again\n\n");
36   return 1;
37  }
38
39  // Allocate the memory block
40  block_ptr = (uint32_t*)malloc(sizeof(uint32_t) * qty_words);
41
42  // Check that the memory was successfully allocated
43  if(block_ptr == NULL)
44  {
45   printf("Not enough memory to allocate requested amount.\n\n");
46   return 1;
47  }
48
49  // Store the length of the block and print the block info for the user
50  block_size = qty_words;
51  printf("Allocated a block starting at address 0x%016lX, %d uint32's long\n\n",(uint64_t)block_ptr, block_size);
52  return 0;
53 }
```

File: Project1/src/dealloc.c

```
1  /*
2   * dealloc.c
3   * @brief frees allocated memory
4   * @author D.Doty
5  */
6
7  /* Includes */
8  #include "dealloc.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t dealloc(char* ignore)
```

```
{
 // Check if we have a block allocated
 if(block_ptr != NULL)
 {
  // Block allocated, free it and set ptr to NULL
  free(block_ptr);
  block_ptr = NULL;
  block_size = 0;
  printf("Block freed\n\n");
  return 0;
 }
 else
 {
  // No block allocated
  printf("No memory allocated to free, use 'allocate' first\n\n");
  return 1;
 }
}
```

File: Project1/src/validate_pattern.c

```
/*
 * validate_pattern.c
 * @brief validates a pseudorandom pattern in a memory range
 * @author D.Doty
 */

/* Includes */
#include "validate_pattern.h"

/* Defines */


/* Global Variables */


/* Private Function Prototypes */


/* Function Definition */
int8_t validate_pattern(char* args)
{
 // Parse arguments
 uint64_t address = 0;
 uint64_t word_qty = 1; // Default. Validates the given address only if no length is given.
 uint64_t seed = 1;  // Default. Uses 1 as seed if not provided (same as generate function).
 io_parse(args, 3, &address, &word_qty, &seed);

 // Check if the provided memory address and word quantity are in allocated block
 // Get user confirmation to proceed if they're out of range
 if(valid_range(address, word_qty) != 0)
 {
  // User has indicated they don't want to proceed. Returns to main for a new command
  return 1;
 }

 // Invert
 printf("Discrepancies will be printed below:\n");
 printf("Word#\tAddress\t\t\tContents S/B\tContents IS\n");
 clock_t begin = clock();
 uint32_t last = xorshift((uint32_t)seed);

 // Loop through the address range specified and compare against PRNG
 for(uint64_t i = 0; i < word_qty*sizeof(uint32_t); i += sizeof(uint32_t))
 {
  // If memory does not match PRNG, print that location
  if(*((uint32_t*)(address + i)) != last)
  {
   printf("%lu\t0x%016lX\t0x%08X\t0x%08X\n",
          i/sizeof(uint32_t),
          address + i,
          last,
          *((uint32_t*)(address + i)));
  }
  // Generate a new PRN for next loop
  last = xorshift(last);
 }

 clock_t end = clock();
 // Calculate execution time in uS to avoid floating point
 uint64_t execution_time = (1000000*(uint64_t)(end - begin))/CLOCKS_PER_SEC;
 printf("\nExecution time: %lu uS\n\n", execution_time);
 return 0;
}
```

File: Project1/src/help.c

```
/*
 * help.c
 * @brief display basic user help messages
 * @author D.Doty
 */

/* Includes */
#include "help.h"

/* Defines */


/* Global Variables */


/* Private Function Prototypes */


/* Function Definition */
void help_welcome()
{
 printf("\nWelcome to Memory Tester\n");
 printf("For command list type 'help'\n\n");
}


int8_t help(char* ignore)
{
 printf("Note command format calls for a space between command and each argument\n");
 printf("E.G. command argument1 argument2\n");
 printf("Arguments can be provided in hex or decimal, hex must be prepended with '0x'\n\n");

 // Loop through the command table printing the commands and help prompts
 printf("Commands:\n");
 for (int i = 0; i < command_quantity; i++)
 {
  printf("%-10s ->\t%s\n", command_human[i], command_help[i]);
```

```
38  }
39  printf("\n");
40  return 0;
41 }
```

File: Project1/src/write_pattern.c

```
1   /*
2    * write_pattern.c
3    * @brief writes a pseudorandom pattern to a memory range
4    * @author D.Doty
5    */
6
7   /* Includes */
8   #include "write_pattern.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t write_pattern(char* args)
21 {
22  // Parse arguments
23  uint64_t address = 0;
24  uint64_t word_qty = 1; // Default. If user does not put value, it assumes they only want to write the provided address
25  uint64_t seed = 1;  // Default. If user does not supply seed, it is assumed to be zero.
26  io_parse(args, 3, &address, &word_qty, &seed);
27
28  // Check if the provided memory address and word quantity are in allocated block
29  // Get user confirmation to proceed if they're out of range
30  if(valid_range(address, word_qty) != 0)
31  {
32   // User has indicated they don't want to proceed. Returns to main for a new command
33   return 1;
34  }
35
36  // Write PRNG Pattern to Memory
37  printf("Writing PRN's...\n");
38  clock_t begin = clock();
39  uint32_t last = xorshift((uint32_t)seed);
40  for(uint64_t i = 0; i < word_qty*sizeof(uint32_t); i += sizeof(uint32_t))
41  {
42   // Looping through provided addresses and writing PRN's
43   *((uint32_t*)(address + i)) = last;
44   // Generate a new PRN for next loop
45   last = xorshift(last);
46  }
47  clock_t end = clock();
48  // Calculating execution time in uS to avoid floating point math
49  uint64_t execution_time = (1000000*(uint64_t)(end - begin))/CLOCKS_PER_SEC;
50  printf("Execution time: %lu uS\n\n",execution_time);
51  return 0;
52 }
```

File: Project1/src/write.c

```
1   /*
2    * write.c
3    * @brief function to write a memory location
4    * @author D.Doty
5    */
6
7   /* Includes */
8   #include "write.h"
9
10 /* Defines */
11
12
13 /* Global Variables */
14
15
16 /* Private Function Prototypes */
17
18
19 /* Function Definition */
20 int8_t write(char* args)
21 {
22  // Parse arguments
23  uint64_t address = 0;
24  uint64_t word = 0;
25  io_parse(args, 2, &address, &word);
26
27  // Check if the provided memory address and word quantity are in allocated block
28  // Get user confirmation to proceed if they're out of range
29  if(valid_range(address, 1) != 0)
30  {
31   // User has indicated they don't want to proceed. Returns to main for a new command
32   return 1;
33  }
34
35  // Write
36  printf("Writing...\n");
37  printf("Address\t\t\tContents Hex\tContents Dec\n");
38  *((uint32_t*)address) = word;
39  printf("0x%016lX\t0x%08X\t%u\n\n", address, *((uint32_t*)address), *((uint32_t*)address));
40  return 0;
41 }
```

File: Project1/src/xorshift.c

```
1   /*
2    * xorshift.c
3    * @brief uses the xorshift PRNG by George Marsaglia
4    * https://en.wikipedia.org/wiki/Xorshift
5    * @author D.Doty
6    */
7
8   /* Includes */
9   #include "xorshift.h"
10
11 /* Defines */
12
13
14 /* Global Variables */
```

```
15
16
17 /* Private Function Prototypes */
18
19
20 /* Function Definition */
21 // Note that this implementation is extremely similar to the one covered in
22 // "Xorshift RNGs" by George Marsaglia
23 // I do not claim this as my original work
24 uint32_t xorshift(uint32_t last_value)
25 {
26  last_value ^= last_value << 13;
27  last_value ^= last_value >> 17;
28  last_value ^= last_value << 5;
29  return last_value;
30 }
```

File: Project1/src/end.c

```
1  /*
2   * end.h
3   * @brief function to end the program
4   * @author D.Doty
5  */
6
7  /* Includes */
8
9  #include "end.h"
10
11 /* Defines */
12
13
14 /* Global Variables */
15
16
17 /* Private Function Prototypes */
18
19
20 /* Function Definition */
21 int8_t end(char* ignore)
22 {
23  exit(0);
24 }
```

File: Project1/src/io.c

```
1    /*
2     * io.h
3     * @brief Input/Output functions
4     * @author D.Doty
5    */
6
7    /* Includes */
8    #include "io.h"
9
10
11   /* Defines */
12
13
14   /* Global Variables */
15
16
17   /* Private Function Prototypes */
18
19
20   /* Function Definition */
21   // Gets user input and matches the first word to a command in the command table
22   // everything after the first whitespace or newline is passed on to the command as arguments
23   // If no command is matched, it returns -1 and writes an error message
24   struct io cmd_get()
25   {
26    // Init output structure
27    struct io output;
28    output.command = -1;
29
30    // Declare and take in input
31    char input[MAX_INPUT_LENGTH];
32    if(!fgets(input, MAX_INPUT_LENGTH, stdin))
33    {
34     exit(1);
35    }
36
37    // Need to add something here that checks if we're testing and prints the input
38    // Since redirected input isn't shown in the terminal it can be confusing
39    printf("\n");
40
41    // Split the input into command and args (splits at first space or newline)
42    char token[MAX_INPUT_LENGTH];  // Token stores the command string
43    token[0] = 0;
44
45    // Loop through input string looking for spaces or newlines
46    for(uint8_t i = 0; i < MAX_INPUT_LENGTH; i++)
47    {
48     // If char is a space or newline
49     if((input[i] == 0x20) || (input[i] == 0x0A))
50     {
51      // Set the space or newline to a null and use copy to move command into token
52      input[i] = 0;
53      strcpy(&token[0],&input[0]);
54      strcpy(output.args,&input[i+1]);
55      break;
56     }
57    }
58
59    // Turn the command token into a command index #
60    // Loop through the command table and compare table to input char by char for a match
61    for(int8_t command_index = 0; command_index < command_quantity; command_index++)
62    {
63     // Loop through the input string and compare char by char
64     uint8_t char_index = 0;
65     while(token[char_index] == command_human[command_index][char_index])
66     {
67      char_index++;
68      // If both the input and the command table show a null, then the whole command matched
69      if((token[char_index] == 0) && (command_human[command_index][char_index] == 0))
70      {
71       output.command = command_index;
72       return output;
73      }
74     }
75    }
76    printf("Invalid command, try 'help'\n\n");
77    return output;
78   }
```

```
79
80  // Turn and argument string into numbers in variables
81  // Variadic function. User provides how many arguments they expect to get
82  // and addresses where they want to store them.
83  // Function splits the string on whitespaces or newlines and turns each chunk
84  // into a number. 0x is converted as hex, otherwise converted as decimal
85  void io_parse(char* arg_string, uint8_t arg_qty, ...)
86  {
87   // Initialize macros for keeping track of variadic arguments
88   va_list arg_ptr;
89   va_start(arg_ptr, arg_qty);
90   uint8_t arg_end;
91
92   // Loop for each expected argument
93   while(arg_qty > 0)
94   {
95    // Take in the pointer to the output variable
96    uint64_t* var_ptr = va_arg(arg_ptr, uint64_t*);
97
98    // Loop through the arg_string to get the first arg
99    arg_end = 0;
100   while(!((arg_string[arg_end] == 0x20) || (arg_string[arg_end] == 0x0A)))
101   {
102    arg_end++;
103    if(arg_end > MAX_INPUT_LENGTH)
104    {
105     return;
106    }
107   }
108   // Drop a null in
109   arg_string[arg_end] = 0;
110
111   // Check for #, indicates special argument
112   if(arg_string[0] == 0x23)
113   {
114    // Need to flesh this out into a whole function for different #'s
115    *var_ptr = (uint64_t)block_ptr;
116   }
117   else
118   {
119    // Turn the first arg into a number and put it in the variable
120    *var_ptr = string_num(arg_string);
121   }
122   // Move the string pointer down to the next part of the list
123   arg_string = &arg_string[arg_end + 1];
124   arg_qty--;
125  }
126
127  // Input ends with a newline then a null. The last thing left in input
128  // should be a null. If it isn't, then we have more arguments than the prog expected
129  if(arg_string[0] != 0)
130  {
131   printf("Too many args or trailing spaces, ignoring extras\n\n");
132  }
133 }
134
135 // Determines if a string is hex or decimal and calls appropriate conversion
136 uint64_t string_num(char* string)
137 {
138  uint64_t result = 0;
139
140  //determine if its hex or decimal and call the appropriate func
141  // if string starts with 0X or 0x its hex
142  if((string[0] == '0') && ((string[1] == 'x')||(string[1] == 'X')))
143  {
144   result = string_hex(&string[2]); //string[2] cuts off the '0x'
145  }
146  else
147  {
148   result = string_dec(string);
149  }
150  return result;
151 }
152
153 // Converts decimal string to number
154 uint64_t string_dec(char* string)
155 {
156  uint8_t length = strlen(string);
157  uint64_t result = 0;
158
159  // Loop through input string, multiplying each number by its 10 power
160  for(int8_t i = length - 1; i >= 0; i--)
161  {
162   // Check if char is outside ASCII range for numbers
163   if(string[i] < 0x30 || string[i] > 0x39)
164   {
165    printf("Argument '%s' was not valid decimal. If you wanted hex, prepend with '0x'\n\n", string);
166    return 0;
167   }
168   result += ((uint8_t)(string[i] - 0x30))*power(10,length -1 - i);
169  }
170  return result;
171 }
172
173 // Converts hex string to number
174 uint64_t string_hex(char* string)
175 {
176  uint8_t length = strlen(string);
177  uint64_t result = 0;
178  uint8_t value = 0;
179
180  // Loop through input string, multiplying each number by its 16 power
181  for(int8_t i = length - 1; i >= 0; i--)
182  {
183   // Number between 0 and 9, convert to number
184   if((string[i] >= 0x30) && (string[i] <= 0x39))
185   {
186    value = (uint8_t)string[i] - 0x30;
187   }
188   // Number between A and F, convert to 10-15
189   else if((string[i] >= 0x41) && (string[i] <= 0x46))
190   {
191    value = ((uint8_t)string[i]) - 0x37;
192   }
193   // Number between a and f, convert to 10-15
194   else if((string[i] >= 0x61) && (string[i] <= 0x66))
195   {
196    value = ((uint8_t)string[i]) - 0x57;
197   }
198   // Invalid character for hex
199   else
200   {
201    printf("Argument '%s' was not valid hex. Remember hex must be prepended with '0x'\n\n", string);
202    return 0;
203   }
204   result += value*power(16,length -1 - i);
205  }
206  return result;
207 }
208
```

```
209 // General integer exponentiation
210 uint64_t power(uint32_t base, uint32_t exponent)
211 {
212  uint64_t result = 1;
213  for(uint8_t i = 0; i < exponent; i++)
214  {
215   result *= base;
216  }
217  return result;
218 }
219
220 // Verify a given address and range are inside allocated memory
221 // If they are not, prompt the user if they want to proceed
222 int8_t valid_range(uint64_t address, uint64_t word_qty)
223 {
224  char response[4] = ""; // Response string
225
226  // Check if memory is allocated
227  if(block_ptr == NULL)
228  {
229   printf("No memory allocated, Proceed? (y/n)\n");
230   fgets(response, 3, stdin);
231  } // Check if given address and word qty fit in allocated memory
232  else if((address < (uint64_t)block_ptr) || ((address + (word_qty - 1)*sizeof(uint32_t)) > ((uint64_t)block_ptr + (block_size - 1)*sizeof(uint32_t))))
233  {
234   printf("Attempting to access memory out of the allocated range, Proceed? (y/n)\n");
235   fgets(response, 3, stdin);
236  }
237
238  // Act according to response
239  if(response[0] == 0)
240  {
241   return 0; // Address and range are acceptable
242  }
243  else if((response[0] == 'y') || (response[0] == 'Y'))
244  {
245   return 0; // Address and range are not acceptable, but user overrides
246  }
247  else
248  {
249   return 1; // Address and range are not acceptable, user acquiesces
250  }
251 }
```

File: Project1/src/main.c

```
1  /*
2   * D.Doty
3   * ECEN 5813
4   * CU Boulder
5   * Fall 2018
6   * Project 1
7   */
8
9
10 /* Includes */
11 #include <stdio.h>
12 #include <stdint.h>
13 #include "commandtable.h"
14 #include "io.h"
15 #include "help.h"
16 #include "end.h"
17 #include "allocate.h"
18 #include "dealloc.h"
19 #include "display.h"
20 #include "write.h"
21 #include "invert.h"
22 #include "write_pattern.h"
23 #include "validate_pattern.h"
24
25
26 /* Defines */
27
28
29
30 /* Function Prototypes */
31
32
33
34 /* Global Variables */
35 uint32_t* block_ptr = NULL;
36 uint32_t block_size = 0;
37
38 /* Main */
39 int main()
40 {
41  // Initialize the command table and fill it
42  command_table_init(9);
43  add_command("help", "returns list of available commands and their function, no arguments", help);
44  add_command("exit", "exits the program, all is lost, no arguments", end);
45  add_command("allocate", "allocates a block of memory,\n\t\targument is number of 32 bit words to allocate", allocate);
46  add_command("free", "frees the currently allocated block of memory, no arguments", dealloc);
47  add_command("display", "displays memory contents at a specified address range,\n\t\targument is start_address and length of block to display", display);
48  add_command("write", "writes provided 32 bit word to specified address,\n\t\targument is memory_address and 32bit word", write);
49  add_command("invert", "inverts contents at a specified address range,\n\t\targument is the memory_address and length of block to invert", invert);
50  add_command("prng", "writes pseudo random 32 bit words to specified address range,\n\t\targument is the memory_address, length of block to write, and prng seed", write_pattern);
51  add_command("validate", "validates pseudo random 32 bit words at specified address range,\n\t\targument is the memory_address, length of block to validate, and prng seed\n\t\tnote this won't work with a different starting
52
53  // Print the welcome message
54  help_welcome();
55
56  // Loop forever getting user commands
57  while(1)
58  {
59   printf(">>");     // Input point marker
60   struct io result = {-1, ""}; // Init the user input structure
61   result = cmd_get();    // Get the user input
62   if(result.command != -1)  // If command is valid, execute it
63   {
64    command_table[result.command](result.args);
65   }
66  }
67 }
```

File: Project1/src/template.c

```
1  /*
2   * temp.c
3   * @brief temp
4   * @author D.Doty
5   */
6
7  /* Includes */
```

```
8
9  #include "template.h"
10
11 /* Defines */
12
13
14 /* Global Variables */
15
16
17 /* Private Function Prototypes */
18
19
20 /* Function Definition */
```

File: Project1/include/dealloc.h

```
/*
* dealloc.h
* @brief frees allocated memory
* @author D.Doty
*/

/* Header Guard */
#ifndef DEALLOC_H
#define DEALLOC_H

/* Includes */
#include
#include
#include

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t dealloc(char* ignore);

#endif //FREE_H
```

File: Project1/include/help.h

```
/*
* help.h
* @brief display help dialog to get user started
* @author D.Doty
*/

/* Header Guard */
#ifndef HELP_H
#define HELP_H

/* Includes */

#include
#include
#include "commandtable.h"

/* Defines */


/* Global Variables */


/* Function Prototypes */

void help_welcome();

int8_t help(char* ignore);

#endif //HELP_H
```

File: Project1/include/commandtable.h

```
/*
* command table.h
* @brief arrays to hold commands and functions to create and interact with them
* @author D.Doty
*/

/* Header Guard */
#ifndef COMMANDTABLE_H
#define COMMANDTABLE_H

/* Includes */
#include
#include
#include
#include

/* Defines */
typedef int8_t (*command_proto)(char*);

/* Global Variables */
extern char** command_human;
extern char** command_help;
extern command_proto* command_table;
extern uint8_t command_quantity;
```

```
/* Global Function Prototypes */

// Creates the empty command table
int8_t command_table_init(uint8_t num_of_commands);

// Adds a command to the command table
int8_t add_command(char* human_name, char* help_msg, command_proto func_pointer);

#endif //COMMANDTABLE_H
```

File: Project1/include/invert.h

```
/*
* invert.h
* @brief inverts a memory range
* @author D.Doty
*/

/* Header Guard */
#ifndef INVERT_H
#define INVERT_H

/* Includes */
#include
#include
#include
#include "io.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t invert(char* args);

#endif //INVERT_H
```

File: Project1/include/template.h

```
/*
* temp.h
* @brief temp
* @author D.Doty
*/

/* Header Guard */
#ifndef TEMP_H
#define TEMP_H

/* Includes */


/* Defines */


/* Global Variables */


/* Global Function Prototypes */


#endif //TEMP_H
```

File: Project1/include/write.h

```
/*
* write.h
* @brief function to write a memory location
* @author D.Doty
*/

/* Header Guard */
#ifndef WRITE_H
#define WRITE_H

/* Includes */
#include
#include
#include "io.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t write(char* args);

#endif //WRITE_H
```

File: Project1/include/validate_pattern.h

```
/*
* validate_pattern.h
* @brief validates a pseudorandom pattern in a memory range
* @author D.Doty
*/

/* Header Guard */
#ifndef VALIDATE_PATTERN_H
#define VALIDATE_PATTERN_H

/* Includes */
#include
#include
#include
#include "io.h"
#include "xorshift.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t validate_pattern(char* args);

#endif //VALIDATE_PATTERN_H
```

File: Project1/include/write_pattern.h

```
/*
* write_pattern.h
* @brief writes a pseudorandom pattern to a memory range
* @author D.Doty
*/

/* Header Guard */
#ifndef WRITE_PATTERN_H
#define WRITE_PATTERN_H

/* Includes */
#include
#include
#include
#include "io.h"
#include "xorshift.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t write_pattern(char* args);

#endif //WRITE_PATTERN_H
```

File: Project1/include/end.h

```
/*
* end.h
* @brief function to end the program
* @author D.Doty
*/

/* Header Guard */
#ifndef END_H
#define END_H

/* Includes */
#include
#include

/* Defines */


/* Global Variables */


/* Global Function Prototypes */

int8_t end(char* ignore);

#endif //END_H
```

File: Project1/include/allocate.h

```
/*
* allocate.h
* @brief command to allocate memory
* @author D.Doty
*/
```

```
/* Header Guard */
#ifndef ALLOCATE_H
#define ALLOCATE_H

/* Includes */
#include
#include
#include
#include "io.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t allocate(char* args);

#endif //ALLOCATE_H
```

File: Project1/include/display.h

```
/*
* display.h
* @brief function to display a memory location
* @author D.Doty
*/

/* Header Guard */
#ifndef DISPLAY_H
#define DISPLAY_H

/* Includes */
#include
#include
#include "io.h"

/* Defines */


/* Global Variables */
extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
int8_t display(char* args);

#endif //DISPLAY_H
```

File: Project1/include/io.h

```
/*
* io.h
* @brief Input/Output functions
* @author D.Doty
*/

/* Header Guard */
#ifndef IO_H
#define IO_H

/* Includes */
#include
#include
#include
#include
#include "commandtable.h"


/* Defines */
#define MAX_INPUT_LENGTH 36

/* Global Variables */
struct io
{
int8_t command;
char args[MAX_INPUT_LENGTH];
};

extern uint32_t* block_ptr;
extern uint32_t block_size;

/* Global Function Prototypes */
struct io cmd_get();

void io_parse(char* arg_string, uint8_t arg_qty, ...);

uint64_t string_num(char* string);

uint64_t string_dec(char* string);

uint64_t string_hex(char* string);

uint64_t power(uint32_t base, uint32_t exponent);

int8_t valid_range(uint64_t address, uint64_t word_qty);
```

```
#endif //IO_H
```

File: Project1/include/xorshift.h

```
/*
* xorshift.h
* @brief uses the xorshift PRNG by George Marsaglia
* https://en.wikipedia.org/wiki/Xorshift
* @author D.Doty
*/

/* Header Guard */
#ifndef XORSHIFT_H
#define XORSHIFT_H

/* Includes */
#include

/* Defines */


/* Global Variables */


/* Global Function Prototypes */
uint32_t xorshift(uint32_t last_value);

#endif //XORSHIFT_H
```

File: Project1/commented_testinput

```
// Print help
help

// Verify everything fails when no memory is allocated
// n says no to override question
display #address
n
write #address 0xFF
n
invert #address
n
prng #address
n
validate #address
n

// Allocation should fail with 0 argument
allocate 0
allocate 10

// Allocating again should fail
allocate 10

// Test all functions in address range
display #address 10

// Write some stuff to the first slot
write #address 0xFF00

// Invert and display slot 1
invert #address
display #address

// Generate PRN's for the whole range
prng #address 10

// Working validation
validate #address 10

// Write an address to force a failure
write #address 0xFFFF
validate #address 10

// Fix the PRN, test with new seed
prng #address 10 200

// Double invert whole range
display #address 10
invert #address 10
validate #address 10 200
invert #address 10
validate #address 10 200

// Free allocated block
free

// Double free should fail
free

// Exit
exit

// Test cases to add:
// Showing all functions prompt for override when used out of range
// Showing all functions work or fail appropriately with no arguments, or too many arguments
```