Help.h

```
#ifndef __Help_h__
#define __Help_h__

#include "Main.h"

#define Number_of_Help_Functions 8

extern const char h[], hd[], h1[], h2[], h3[], h4[], h5[], h6[], h7[], h8[], h9[], hre[], hr1[], hr2[], hr3[];

typedef struct
{
char ht1[5];
char ht2[9];
char ht3[8];
char ht4[9];
char ht5[8];
char ht6[7];
char ht7[11];
char ht8[14];
```

```
}Help;

extern Help help, *help_ptr;
extern char *help_ptr2, help_check[20], help_print[500];
extern uint8_t help_j, help_k;
extern void (*Help_Func_Ptr[8]) (void);

void Help_Init(void);
void Help_Display(void);
void Help_Lookup(void);

void Help_Response1(void);
void Help_Response2(void);
void Help_Response3(void);
void Help_Response4(void);
void Help_Response5(void);
void Help_Response6(void);
void Help_Response7(void);
```

```
void Help_Response8(void);

#endif
```

Input.h

```
#ifndef __Input_h__
#define __Input_h__

#include "Main.h"
#include "Help.h"
#include "Memalloc.h"
#include "Memfree.h"
#include "Memwrite.h"
#include "Memread.h"
#include "Meminv.h"
#include "Patterngen.h"

#define Number_of_Input_Functions 9

extern const char ire[], i1[], i2[], i3[], i4[], i5[], i6[], i7[], i8[], i9[];
extern void (*Input_Func_Pointer[9]) (void);
```

```
extern void (*Input_Space_Func_Pointer[9]) (void);

typedef struct
{
char it1[5];
char it2[5];
char it3[9];
char it4[8];
char it5[9];
char it6[8];
char it7[7];
char it8[11];
char it9[14];
}Input;

extern Input input_search, *input_ptr;
extern char *input_ptr2, input_check[20], error_flag, hex_flag;
extern uint8_t input_j, input_k;

void Command_Error(void);
```

/*void Help_Display(void);
void Help_Lookup(void);*/
void Exit_Func(void);
void Memalloc_Func(void);
void Space_Memalloc_Func(void);
void Memfree_Func(void);
void Memwrite_Func(void);
void Space_Memwrite_Func(void);
void Memread_Func(void);
void Space_Memread_Func(void);
void Meminv_Func(void);
void Space_Meminv_Func(void);
void Patterngen_Func(void);
void Space_Patterngen_Func(void);
void Patternverify_Func(void);
void Space_Patternverify_Func(void);
void Input_Init(void);
void Input_Cleanup(void);

```
void Valid_Integer_Input(void);
void String_to_Decimal(char *stod_ptr);
void String_to_Hex(char *stox_ptr);
void Input_Lookup(void);


#endif
```

Main.h

```
#ifndef __Main_h__
#define __Main_h__

#include<stdint.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>

#include "Config.h"

```

```
#define Invalid() printf("\nInvalid Command\n");

extern char input[250], input1[50], input2[50], input3[50], input4[50], input5[50];
extern uint8_t main_i, main_j, exit_flag, space_flag, relative_address;
extern uint32_t value, value1, value2, value3, value4;

extern char m_print[50];
extern uint8_t print;

extern clock_t t;

extern uint32_t *mem_ptr, *mem_ptr2, mem_max, *mem_original;

void Array_Cleanup(char *clean_ptr);
void Detailed_Output(void);
void clkbegin(void);
void clkend(void);
```

```
#endif
```

Memalloc.h

```
#ifndef __Memalloc_h__
#define __Memalloc_h__

#include "Main.h"

extern uint32_t mac_i;

//void mac_init(void)
void mem_clear(void);
void memalloc(void);

#endif
```

Memfree.h

```
#ifndef __Memfree_h__
#define __Memfree_h__

#include "Main.h"
```

```
void memfree(void);

#endif
```

Meminv.h

```
#ifndef __Meminv_h__
#define __Meminv_h__

#include "Main.h"

void meminv(void);

#endif
```

Memread.h

```
#ifndef __Memread_h__
#define __Memread_h__

#include "Main.h"

void memread(void);
```

```
#endif
```

Memwrite.h

```
#ifndef __Memwrite_h__
#define __Memwrite_h__

#include "Main.h"

void memwrite(void);

#endif
```

Patterngen.h

```
#ifndef __Patterngen_h__
#define __Patterngen_h__

#include "Main.h"

extern float random, seed;
extern uint32_t max, range, random_value, *pattern_original;

```

```
void generator(void);
void patterngen(void);


#endif
```

Help.c

```
#include "Help.h"

const char h[] = "help";
const char hd[] = "\n\nDirections to use help command:\nPut a whitespace after help and then type the command\nTo get information on using that command\nFor example - type: help exit\n\nList of valid Commands:\n";
const char h1[] = "exit";
const char h2[] = "memalloc";
const char h3[] = "memfree";
const char h4[] = "memwrite";
const char h5[] = "memread";
const char h6[] = "meminv";
const char h7[] = "patterngen";
const char h8[] = "patternverify";
const char hre[] = "\nCommand not recognized\n";
const char hr1[] = "\nexit: \nType exit to close the program\n";
```

```
const char hr2[] = "\nMemory Allocation:\n\nType memalloc and then enter the\nnumber of memory locations that you want\nto use and have access to.\n\nAlternatively, type memalloc <number> without <> for value\n";

const char hr3[] = "\nMemory Free:\nType memfree to release the \npreviously allocated memory locations\n";

const char hr4[] = "\nMemory Write:\nType memwrite and then enter the address and data\nto write 32bit data at memory location of your choice.\n\nAlternatively, type memwrite <address> <data> without <> for values\n";

const char hr5[] = "\nMemory Read:\nType memread to read 32bit data in hex\nat memory location of your choice.\n\nAlternatively, type memread <address> without <> for value\n";

const char hr6[] = "\nMemory Inverse:\nType meminv to invert all bits of a\n32bit memory block at location of your choice\n";

const char hr7[] = "\nPsuedo Random Pattern Generation:\nType patterngen and then enter maximum value,\nseed, number of 32bits words that you wish to generate,\nand starting memory location to store the pattern\nto generate multiple psuedo random numbers.\n\nAlternatively, type patterngen <starting memory address> <length of pattern> <seed> <maxvalue>\nwithout <> for values\n";

const char hr8[] = "\nPsuedo Random Pattern Verification:\nType patternverify and then enter maximum value,\nseed, number of 32bits words that you wish to verify,\nand starting memory location pointing to the stored pattern\nto verify multiple psuedo random numbers.\n\nAlternatively, type patternverify <starting memory address> <length of pattern> <seed> <maxvalue>\nwithout <> for values\n";

Help help, *help_ptr;

char *help_ptr2, help_check[20], help_print[500];

uint8_t help_j, help_k;
```

```
void (*Help_Func_Ptr[8]) (void) =
{
Help_Response1,
Help_Response2,
Help_Response3,
Help_Response4,
Help_Response5,
Help_Response6,
Help_Response7,
Help_Response8
};

void Help_Response1(void)
{
printf("%s",hr1);
}

void Help_Response2(void)
{
printf("%s",hr2);
```

```
}

void Help_Response3(void)
{
printf("%s",hr3);
}

void Help_Response4(void)
{
printf("%s",hr4);
}

void Help_Response5(void)
{
printf("%s",hr5);
}
void Help_Response6(void)
{
printf("%s",hr6);
}
```

```
void Help_Response7(void)
{
printf("%s",hr7);
}

void Help_Response8(void)
{
printf("%s",hr8);
}

void Help_Init(void)
{
help_ptr = &help;
help_ptr2 = (char *)&help_ptr->ht1;
strcpy(help.ht1, h1);
strcpy(help.ht2, h2);
strcpy(help.ht3, h3);
strcpy(help.ht4, h4);
strcpy(help.ht5, h5);
```

```
strcpy(help.ht6, h6);
strcpy(help.ht7, h7);
strcpy(help.ht8, h8);
}

void Help_Display(void)
{
printf("%s",hd);
printf("\n\t%s",h);
printf("\n\t%s",h1);
printf("\n\t%s",h2);
printf("\n\t%s",h3);
printf("\n\t%s",h4);
printf("\n\t%s",h5);
printf("\n\t%s",h6);
printf("\n\t%s",h7);
printf("\n\t%s",h8);
printf("\n");
}
```

```
void Help_Lookup(void)
{
help_ptr2 = (char *)&help_ptr->ht1;
Array_Cleanup(help_check);
for(help_k = 0; help_k < Number_of_Help_Functions; help_k
++)//Number_of_Help_Functions
{
for(help_j = 0;; help_j++)
{
help_check[help_j] = *help_ptr2;
help_ptr2 += 1;
if(help_check[help_j] == 0)
{
break;
}
}
if(strcmp(help_check, input2) == 0)
{
(*Help_Func_Ptr[help_k])();
break;
```

```
}
else
{
if(help_k == (Number_of_Help_Functions - 1))
{
printf("%s",hre);
}
}
Array_Cleanup(help_check);
}
}
```

Input.c

```
#include "Input.h"

const char ire[] = "\nCommand not recognized\n";
const char i1[] = "help";
const char i2[] = "exit";
const char i3[] = "memalloc";
const char i4[] = "memfree";
```

```
const char i5[] = "memwrite";
const char i6[] = "memread";
const char i7[] = "meminv";
const char i8[] = "patterngen";
const char i9[] = "patternverify";

Input input_search, *input_ptr;
char *input_ptr2, input_check[20], error_flag, hex_flag;
char input[250], input1[50], input2[50], input3[50], input4[50], input5[50];
uint8_t input_j, input_k, exit_flag, power;
uint32_t value, value1, value2, value3, value4;

void (*Input_Func_Pointer[9]) (void) =
{
Help_Display,
Exit_Func,
Memalloc_Func,
Memfree_Func,
Memwrite_Func,
Memread_Func,
```

```
Meminv_Func,
Patterngen_Func,
Patternverify_Func
};

void (*Input_Space_Func_Pointer[9]) (void) =
{
Help_Lookup,
Command_Error,
Space_Memalloc_Func,
Command_Error,
Space_Memwrite_Func,
Space_Memread_Func,
Space_Meminv_Func,
Space_Patterngen_Func,
Space_Patternverify_Func
};

void Command_Error(void)
{
```

```
printf("%s",ire);
}

void Exit_Func(void)
{
exit_flag = 1;
}

void Memalloc_Func(void)
{
printf("\nEnter number of 32bit words for malloc: ");
hex_flag = 0;
Valid_Integer_Input();
if(error_flag == 0)
{
value1 = value;
memalloc();
}
else
{
```

```
value1 = 0;
}
printf("\n%d Blocks have been allocated\n", value1);
}

void Space_Memalloc_Func(void)
{
if(space_flag == 1)
{
String_to_Decimal(input2);
if(error_flag == 0)
{
value1 = value;
memalloc();
}
}
else
{
value = 0;
Command_Error();
```

```
}
printf("\n%d Blocks have been allocated\n", value);
}

void Memfree_Func(void)
{
memfree();
}

void Memwrite_Func(void)
{
if(relative_address)
{
printf("\nEnter the relative address of the memory location: ");
hex_flag = 0;
}
else
{
printf("\nEnter the absolute address of the memory location: ");
hex_flag = 1;
```

```
}
Valid_Integer_Input();
if(error_flag == 0)
{
value1 = value;
printf("\nEnter 32bit Data in Hex: ");
hex_flag = 1;
Valid_Integer_Input();
if(error_flag == 0)
{
value2 = 0x00000000;
value2 += value;
memwrite();
}
}
}

void Space_Memwrite_Func(void)
{
if(space_flag == 2)
```

```
{
if(relative_address)
{
String_to_Decimal(input2);
}
else
{
String_to_Hex(input2);
}
if(error_flag == 0)
{
value1 = value;
String_to_Hex(input3);
if(error_flag == 0)
{
value2 = value;
memwrite();
}
}
```

```
}
else
{
value = 0;
Command_Error();
}
}

void Memread_Func(void)
{
if(relative_address)
{
printf("\nEnter the relative address of the memory location: ");
hex_flag = 0;
}
else
{
printf("\nEnter the absolute address of the memory location: ");
hex_flag = 1;
}
```

```
Valid_Integer_Input();
if(error_flag == 0)
{
value1 = value;
memread();
}
}

void Space_Memread_Func(void)
{
if(space_flag == 1)
{
if(relative_address)
{
String_to_Decimal(input2);
}
else
{
String_to_Hex(input2);
}
```

```
if(error_flag == 0)
{
value1 = value;
memread();
}
}
else
{
value = 0;
Command_Error();
}
}

void Meminv_Func(void)
{
if(relative_address)
{
printf("\nEnter the relative address of the memory location: ");
hex_flag = 0;
}
```

```
else
{
printf("\nEnter the absolute address of the memory location: ");
hex_flag = 1;
}
Valid_Integer_Input();
if(error_flag == 0)
{
value1 = value;
meminv();
}
}
void Space_Meminv_Func(void)
{
if(space_flag == 1)
{
if(relative_address)
{
String_to_Decimal(input2);
```

```
}
else
{
String_to_Hex(input2);
}
if(error_flag == 0)
{
value1 = value;
meminv();
}
}
else
{
value = 0;
Command_Error();
}
}

void Patterngen_Func(void)
{
```

```
if(relative_address)
{
printf("\nEnter the relative starting address of the memory location: ");
hex_flag = 0;
}
else
{
printf("\nEnter the absolute starting address of the memory location: ");
hex_flag = 1;
}
Valid_Integer_Input();
if(error_flag == 0)
{
value1 = value;
printf("\nEnter the number of 32bit numbers that you wish to generate: ");
hex_flag = 0;
Valid_Integer_Input();
if(error_flag == 0)
{
```

```
value2 = value;
printf("\nEnter the Seed value of your choice: ");
hex_flag = 0;
Valid_Integer_Input();
if(error_flag == 0)
{
value3 = value;
printf("\nEnter the Maximum value of generated psuedo random number(s): ");
hex_flag = 0;
Valid_Integer_Input();
if(error_flag == 0)
{
value4 = value;
patterngen();
}
}
}
}
}
```

```
void Space_Patterngen_Func(void)
{
if(space_flag == 4)
{
if(relative_address)
{
String_to_Decimal(input2);
}
else
{
String_to_Hex(input2);
}
if(error_flag == 0)
{
value1 = value;
String_to_Decimal(input3);
if(error_flag == 0)
{
value2 = value;
```

```
String_to_Decimal(input4);
if(error_flag == 0)
{
value3 = value;
String_to_Decimal(input5);
if(error_flag == 0)
{
value4 = value;
patterngen();
}
}
}
}
}
else
{
value = 0;
Command_Error();
}
}
```

```
void Patternverify_Func(void)
{
if(relative_address)
{
printf("\nEnter the relative starting address of the memory location: ");
hex_flag = 0;
}
else
{
printf("\nEnter the absolute starting address of the memory location: ");
hex_flag = 1;
}
Valid_Integer_Input();
//if error = 0
value1 = value;
printf("\nEnter the number of 32bit numbers that you wish to verify: ");
hex_flag = 0;
Valid_Integer_Input();
//if error = 0
```

```
value2 = value;
printf("\nEnter the original Seed value: ");
hex_flag = 0;
Valid_Integer_Input();
//if error = 0
value3 = value;
printf("\nEnter the Maximum value of generated psuedo random number(s): ");
hex_flag = 0;
Valid_Integer_Input();
//if error = 0
value4 = value;
//if error = 0, call patternverify
}
void Space_Patternverify_Func(void)
{
if(space_flag == 4)
{
if(relative_address)
{
String_to_Decimal(input2);
```

```
}
else
{
String_to_Hex(input2);
}
//if error = 0
value1 = value;
String_to_Decimal(input3);
//if error = 0
value2 = value;
String_to_Decimal(input4);
//if error = 0
value3 = value;
String_to_Decimal(input5);
//if error = 0
value4 = value;
//if error = 0. call patternverify
}
else
```

```
{
value = 0;
Command_Error();
}
}

void String_to_Decimal(char *stod_ptr)
{
char *stod_i;
stod_i = stod_ptr;
for(; *stod_ptr != 0; stod_ptr ++)
{
if(isdigit(*stod_ptr) == 0)
{
printf("\nNon Integer Value Entered\n");
error_flag = 1;
value = 0;
break;
}
}
```

```
if(*stod_ptr == 0)
{
error_flag = 0;
value = atoi(stod_i);
}
}

void String_to_Hex(char *stox_ptr)
{
char *stox_i;
stox_i = stox_ptr;
while(*stox_ptr != 0)
{
if(((*stox_ptr >= '0') && (*stox_ptr <= '9')) || ((*stox_ptr >= 'a') && (*stox_ptr <= 'f')) ||
((*stox_ptr >= 'A') && (*stox_ptr <= 'F')))
{
stox_ptr += 1;
}
else
{
```

```
break;
}
}
if(*stox_ptr == 0)
{
error_flag = 0;
value = (uint32_t) strtol(stox_i, NULL, 16);
}
else
{
printf("\nNon Hex Value Entered\n");
error_flag = 1;
value = 0;
Command_Error();
}
}

void Valid_Integer_Input(void)
{
char test[50];
```

```
fgets(test, 50, stdin);
input_j = 0;
while(test[input_j] != '\n')
{
input_j += 1;
}
test[input_j] = 0;
if(hex_flag == 0)
{
String_to_Decimal(test);
}
else
{
String_to_Hex(test);
}
}

void Input_Init(void)
{
input_ptr = &input_search;
```

```
input_ptr2 = (char *)&input_ptr->it1;
strcpy(input_search.it1, i1);
strcpy(input_search.it2, i2);
strcpy(input_search.it3, i3);
strcpy(input_search.it4, i4);
strcpy(input_search.it5, i5);
strcpy(input_search.it6, i6);
strcpy(input_search.it7, i7);
strcpy(input_search.it8, i8);
strcpy(input_search.it9, i9);
}

void Input_Cleanup(void)
{
hex_flag = 0;
error_flag = 0;
value = 0;
value1 = 0;
value2 = 0;
value3 = 0;
```

```
value4 = 0;
input_j = 0;
input_ptr2 = (char *)&input_ptr->it1;
Array_Cleanup(input_check);
}

void Input_Lookup(void)
{
Input_Cleanup();
//In the input, entering space after command as a mistake is acceptable, but entering anything after that space is not
if((space_flag != 0) && (input2[0] != 0))
{
for(input_k = 0; input_k < Number_of_Input_Functions; input_k ++)
{
for(input_j = 0;; input_j++)
{
input_check[input_j] = *input_ptr2;
input_ptr2 += 1;
if(input_check[input_j] == 0)
```

```
{
break;
}
}
if(strcmp(input_check, input1) == 0)
{
(*Input_Space_Func_Pointer[input_k])();
break;
}
else
{
if(input_k == (Number_of_Input_Functions - 1))
{
printf("%s",ire);
}
}
input_j = 0;
Array_Cleanup(input_check);
}
}
```

```
else
{
for(input_k = 0; input_k < Number_of_Input_Functions; input_k ++)
{
for(input_j = 0;; input_j++)
{
input_check[input_j] = *input_ptr2;
input_ptr2 += 1;
if(input_check[input_j] == 0)
{
break;
}
}
if(strcmp(input_check, input1) == 0)
{
(*Input_Func_Pointer[input_k])();
break;
}
else
```

```
{
if(input_k == (Number_of_Input_Functions - 1))
{
printf("%s",ire);
}
}
Array_Cleanup(input_check);
}
}
}
```

Main.c

```
#include "Main.h"
#include "Help.h"
#include "Input.h"

char input[250], input1[50], input2[50], input3[50], input4[50], input5[50];
uint8_t main_i, main_j, exit_flag, space_flag, relative_address;

char m_print[50];
uint8_t print = 0;
```

```
clock_t t;

void clkbegin(void)
{
t = clock();
}

void clkend(void)
{
t = clock() - t;
double time_taken = (((double)t)/CLOCKS_PER_SEC) * 1000; // in milli seconds
printf("\nThe process took %f milli seconds to execute: ", time_taken);
}

void Detailed_Output(void)
{
while(1)
{
printf("Do you want to use Detailed information?\n");
```

```
printf("\n Type Y or y to accept, type N or n to reject: ");
fgets(m_print, 50, stdin);
if((m_print[0] == 'Y') || (m_print[0] == 'y'))
{
print = 1;
break;
}
else if((m_print[0] == 'N') || (m_print[0] == 'n'))
{
print = 0;
break;
}
else
{
printf("\nInvalid Input, Try again\n");
}
}
}

void Array_Cleanup(char *clean_ptr)
```

```
{
while(*clean_ptr != 0)
{
*clean_ptr = 0;
clean_ptr += 1;
}
}

int main(void)
{
relative_address = 1;
Array_Cleanup(input);
Array_Cleanup(input1);
Array_Cleanup(input2);
Array_Cleanup(input3);
Array_Cleanup(input4);
Array_Cleanup(input5);
Help_Init();
Input_Init();
// mac_init();
```

```
char address_type[50];
while(1)
{
printf("Do you want to use Relative/Easy Addressing?\n");
printf("\n Type Y or y to accept, type N or n to reject\nand use absolute/direct addressing:
");
fgets(address_type, 50, stdin);
if((address_type[0] == 'Y') || (address_type[0] == 'y'))
{
relative_address = 1;
break;
}
else if((address_type[0] == 'N') || (address_type[0] == 'n'))
{
relative_address = 0;
break;
}
else
{
printf("\nInvalid Input, Try again\n");
```

```
}
}
printf("\nRelative Addressing Value set to: %d\n", relative_address);
while(1)
{
exit_flag = 0;
space_flag = 0;
printf("\nEnter Command: ");
fgets(input, 250, stdin);
main_i = 0;
while(input[main_i] != '\n')
{
main_i += 1;
}
input[main_i] = 0;
main_i = 0;
while((input[main_i] != 0) && (input[main_i] != ' '))
{
input1[main_i] = input[main_i];
main_i += 1;
```

```
}
main_j = 0;
while(input[main_i] != 0)
{
if(input[main_i] == ' ')
{
space_flag += 1;
main_i += 1;
while((input[main_i] != ' ') && (input[main_i] != 0))
{
input2[main_j] = input[main_i];
main_i += 1;
main_j += 1;
}
if(input[main_i] == ' ')
{
space_flag += 1;
main_j = 0;
main_i += 1;
```

```
while((input[main_i] != ' ') && (input[main_i] != 0))
{
input3[main_j] = input[main_i];
main_i += 1;
main_j += 1;
}
if(input[main_i] == ' ')
{
space_flag += 1;
main_j = 0;
main_i += 1;
while((input[main_i] != ' ') && (input[main_i] != 0))
{
input4[main_j] = input[main_i];
main_i += 1;
main_j += 1;
}
if(input[main_i] == ' ')
{
space_flag += 1;
```

```
main_j = 0;
main_i += 1;
while((input[main_i] != ' ') && (input[main_i] != 0))
{
input5[main_j] = input[main_i];
main_i += 1;
main_j += 1;
}
}
}
}
}
main_i += 1;
}
main_i = 0;
main_j = 0;
if(input[main_i] != 0)
{
Input_Lookup();
```

```
if(exit_flag)
{
return 0;
}
}
Array_Cleanup(input);
Array_Cleanup(input1);
Array_Cleanup(input2);
Array_Cleanup(input3);
Array_Cleanup(input4);
Array_Cleanup(input5);
// printf("\n%s\n%s\n%s\n%s",input2,input3,input4,input5);
/* break;
if(input2[main_i] == ' ')
{
space_flag = 1;
main_j = 1;
Input_Lookup();
if(exit_flag)
{
```

```
return 0;
}
}
else
{
while(input[main_i] != 0)
{
compare[main_i] = input[main_i];
main_i += 1;
}
Input_Lookup();
if(exit_flag)
{
return 0;
}
}
Array_Cleanup(input);
Array_Cleanup(input2);
Array_Cleanup(compare);*/
```

```
}

}
```

Memalloc.c

```
#include "Memalloc.h"

uint32_t *mem_ptr, mac_i, mem_max, *mem_original;

void mem_clear(void)
{
for(mac_i = 0; mac_i < value1; mac_i ++)
{
*mem_ptr = 0x00000000;
}
}

void memalloc(void)
{
mem_ptr = (uint32_t *) malloc (4*value1);
mem_original = mem_ptr;
```

```
mem_max = value1;
mem_clear();
Detailed_Output();
if(print)
{
printf("\nThe allocated addresses are as below:\n");
for(mac_i = 0; mac_i < mem_max; mac_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x", mac_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
printf("\n\nThanks for executing the memory allocation operation...\n");
}
```

Memfree.c

```
#include "Memfree.h"

void memfree(void)
{
```

```
if(mem_ptr)
{
free(mem_ptr);
printf("\nThe allocated memory has been successfully freed...\n");
printf("\nThanks for freeing the allocated memory...\n");
}
else
{
printf("\nUnfortunately, you have not been allocated any memory so far and so no memory was freed...\n");
}
}
```

Meminv.c

```
#include "Meminv.h"

uint32_t *mem_ptr2, mem_i;

clock_t t;

void meminv(void)
```

```
{
if(mem_ptr)
{
Detailed_Output();
if(print)
{
printf("\nInformation BEFORE Inverting operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
if(relative_address)
{
value1 = (value1 * 4) + (uint32_t)mem_ptr;
}
mem_ptr2 = (uint32_t *) value1;
```

```
if(print == 0)
{
printf("\nThe data in hex at the specified memory location BEFORE Inverting is %x\n", *mem_ptr2);
}
clkbegin();
*mem_ptr2 ^= 0xFFFFFFFF;
clkend();
if(print == 0)
{
printf("\nThe data in hex at the specified memory location AFTER Inverting is %x\n", *mem_ptr2);
}
else
{
printf("\nInformation AFTER Inverting operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
```

```
}
}
mem_ptr = mem_original;
printf("\nThanks for performing an XOR operation at an allocated memory location...\n");
}
else
{
printf("\nUnfortunately, you have not been allocated any memory so far and so XORing is not possible at this moment...\n");
}
}
```

Memread.c

```
#include "Memread.h"

uint32_t *mem_ptr2, mem_i;

void memread(void)
{
if(mem_ptr)
{
```

```
Detailed_Output();
if(print)
{
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
if(relative_address)
{
value1 = (value1 * 4) + (uint32_t)mem_ptr;
}
mem_ptr2 = (uint32_t *) value1;
if(print == 0)
{
printf("\nThe hex data at the specified location %x is %x\n", mem_ptr2, *mem_ptr2);
}
```

```
printf("\nThanks for reading from an allocated memory location...\n");
}
else
{
printf("\nUnfortunately, you have not been allocated any memory so far and so no memory can be read...\n");
}
}
```

Memwrite.c

```
#include "Memwrite.h"

uint32_t *mem_ptr2, mem_i;

void memwrite(void)
{
if(mem_ptr)
{
Detailed_Output();
if(print)
{
```

```
printf("\nInformation BEFORE write operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
if(relative_address)
{
value1 = (value1 * 4) + (uint32_t)mem_ptr;
}
mem_ptr2 = (uint32_t *) value1;
if(print == 0)
{
printf("\nThe hex data at the specified location BEFORE writing %x is %x\n", mem_ptr2, *mem_ptr2);
}
*mem_ptr2 = value2;
if(print == 0)
```

```
{
printf("\nThe hex data at the specified location AFTER writing %x is %x\n", mem_ptr2, *mem_ptr2);
}
else
{
printf("\nInformation AFTER write operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
printf("\nThanks for writing to an allocated memory location...\n");
}
else
{
printf("\nUnfortunately, you have not been allocated any memory so far and so no memory write cannot be done...\n");
```

```
}

}
```

Patterngen.c

```
#include "Patterngen.h"

//add, len, seed, max

uint32_t *mem_ptr2, mem_i;

float random_number, seed;

uint32_t max, range, random_value, *pattern_original;

clock_t t;

void generator(void)

{

random_number = ((0.4353491074*seed) + 0.8173946121);

while(random_number > 1)

{

random_number /= 10;

}

random_number *= max;
```

```
random_value = (uint32_t)random_number;
printf("\n%x", random_value);
*mem_ptr2 = random_value;
seed = random_number;
}

void patterngen(void)
{
if(mem_ptr)
{
range = value2;
seed = (float) value3;
max = value4;
Detailed_Output();
clkbegin();
if(print)
{
printf("\nInformation BEFORE pattern generate operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
```

```
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
if(relative_address)
{
value1 = (value1 * 4) + (uint32_t)mem_ptr;
}
mem_ptr2 = (uint32_t *) value1;
pattern_original = mem_ptr2;
while(seed > 1)
{
seed /= 10;
}
printf("\nGenerated Psuedo Random Numbers: \n");
uint8_t counter;
for(counter = 0; counter < range; counter ++, mem_ptr2 ++)
{
```

```
generator();
}
mem_ptr2 = pattern_original;
if(print)
{
printf("\nInformation AFTER pattern generate operation\n\n");
printf("\nThe allocated addresses are as below:\n");
for(mem_i = 0; mem_i < mem_max; mem_i ++, mem_ptr ++)
{
printf("\nRelative address: %d \t\t Actual address: %x \t\t Existing hex data at this location in Hex: %x\n", mem_i, mem_ptr, *mem_ptr);
}
}
mem_ptr = mem_original;
clkend();
printf("\nThanks for generating a psuedo random number...\n");
}
else
{
printf("\nUnfortunately, you have not been allocated any memory so far and so no memory write cannot be done...\n");
```

```
}


}
```