PES Project 3 Readme

Jack Campbell

Description

This repo contains custom sources and makefiles for Project 3 as well as adapted and generated code from MCUXpresso and the KL25Z SDK.

A global makefile lives at the project root that contains all the build targets: fb_run, fb_debug, pc_run, pc_debug, and clean. The *_run variants go into the `Release` directory and execute release-specific rules and make logic, whereas the *_debug variants do the same thing in the `Debug` subdirectory.

The configuration specific KL25Z `*.mk` files were all generated by the IDE and saved and checked in so the project could be loaded consistently from the repo. The PC makefiles and project-level makefiles were written by me.

Installation/Execution Notes

These are the steps to build the project in MCUXpresso.

1. Clone the repo
2. In MCUXpresso, click `New > Project`.
3. Select `Makefile project with existing code...`
4. Unselect C++, enter a project name, browse to the directory of the repo, and select `NXP MCU Tools`, then hit next.
5. Now the project is active in the IDE.

Adding build targets

To add the build targets specified in the makefile:

1. Open the Build Targets side window
2. Right click the project name
3. Select New
4. Enter the name of one of the build targets (fb_run, fb_debug, pc_run, pc_debug, and clean)
5. Repeat for all build targets
6. Double clicking on a target name will invoke that target

Running the PC builds

After building the targets (either individually using the build target steps above or just using the hammer icon to build "all"), you can add configurations to run the pc builds in the editor.

1. Right click on the project name in the file hierarchy, select `Run as > Run configurations...`
2. Select `C/C++ Application`
3. Hit `New launch configuration`
4. Select a name for the output configuration (you need one for both Release and Debug)

5. Set the `C/C++ Application` field to the binary you want to run, either `Debug/output/pc_debug` for Debug or `Release/output/pc_run` for Release
6. Hit Apply
7. Hit Run
8. The program should run in the console below.

Running the FB builds

1. Right click on the project name in the file hierarchy, select `Debug as > Debug configurations...`
2. Select `GDB PEMicro Interface Debugging`
3. Hit `New launch configuration`
4. Select a name for the output configuration (you need one for both Release and Debug)
5. Set the `C/C++ Application` field to the binary you want to run, either `Debug/output/kl25z_debug.axf` for Debug or `Release/output/kl25z_run.axf` for Release
6. Hit Apply
7. Hit Debug
8. The program should run in the console below, provided the board is connected successfully.

Observations

A difficult part of this assignment was, again, just getting the makefiles to work nicely for all the different configurations. Adding the alternate unit test executable proved harder than I thought.

I also had trouble getting the LED GPIO to be set up correctly. My code was identical to my other projects, but this was the first time I had to try and mess around with the pin tool in MCUXpresso.

Writing unit tests first saved a lot of time.

Code

```
#include "uCUnit.h"
#include "memory_tests.h"
#include "mem_types.h"
#include "logger.h"
#define ALLOC_SIZE 16

int main()
{
#ifdef DEBUG
        log_enable();
#endif

        {
            UCUNIT_TestcaseBegin("allocate words");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
            for(int i = 0; i < ALLOC_SIZE; i++)
            {
                UCUNIT_CheckIsEqual(((uint8_t*)ptr)[i], 0x0);
            }
```

```
            UCUNIT_CheckIsNotNull(ptr);
            UCUNIT_TestcaseEnd();

            // clean up
            free_words(ptr);
        }

        {
            UCUNIT_TestcaseBegin("allocate words fail");
            uint32_t* ptr = allocate_words(0);
            UCUNIT_CheckIsNull(ptr);
            UCUNIT_TestcaseEnd();

            // clean up
                free_words(ptr);
        }

        {
                UCUNIT_TestcaseBegin("get address");
                uint32_t* ptr = allocate_words(ALLOC_SIZE);
                uint32_t * addr = get_address(ptr, 8);
                UCUNIT_CheckIsEqual(addr, (uint32_t*)((uint8_t*)(ptr) +
8));
                UCUNIT_TestcaseEnd();

                // clean up
                free_words(ptr);
        }

        {
            UCUNIT_TestcaseBegin("free words");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
                UCUNIT_CheckIsEqual(free_words(ptr), SUCCESS);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("free words null");
                UCUNIT_CheckIsEqual(free_words(NULL), FAILED);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("free words garbage");
                UCUNIT_CheckIsEqual(free_words((uint32_t*)0xDEADBEEF),
FAILED);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("write pattern");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
                const uint8_t seed = 2;
```

```
                mem_status status = write_pattern(ptr, ALLOC_SIZE, seed);
                UCUNIT_CheckIsEqual(status, SUCCESS);

                // clean up
                free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("display memory");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
                uint8_t * disp = display_memory(ptr, ALLOC_SIZE);
                UCUNIT_CheckIsNotNull(disp);
                // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("verify pattern");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
            const int8_t seed = 2;

            mem_status status = write_pattern(ptr, ALLOC_SIZE, seed);
            UCUNIT_CheckIsEqual(status, SUCCESS);

            uint32_t * nonMatchingValues = verify_pattern(ptr, ALLOC_SIZE,
seed);
            UCUNIT_CheckIsNull(nonMatchingValues);

            // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("write memory");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
            for(int i = 0; i < ALLOC_SIZE/sizeof(uint32_t); i++)
            {
                        ptr[i] = 0xABABABAB;
                }

                uint8_t* ptrBytes = (uint8_t*)ptr;

                uint32_t offset = 2;
            uint8_t valToWrite = 0xEE;
            mem_status status = write_memory(ptr+offset, valToWrite);

            for(int i = 0; i < ALLOC_SIZE; i++)
            {
                UCUNIT_CheckIsEqual(ptrBytes[i], i == 8 ? 0xEE : 0xAB);
            }
```

```
            UCUNIT_CheckIsEqual(status, SUCCESS);

            // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("write memory fail");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
            for(int i = 0; i < ALLOC_SIZE/sizeof(uint32_t); i++)
            {
                ptr[i] = 0xABABABAB;
            }

            uint8_t* ptrBytes = (uint8_t*)ptr;

            uint32_t offset = 200;
            uint16_t valToWrite = 0xFFEE;
            mem_status status = write_memory(get_address(ptr, offset),
valToWrite);
            UCUNIT_CheckIsEqual(status, FAILED);

            for(int i = 0; i < ALLOC_SIZE; i++)
            {
                UCUNIT_CheckIsEqual(ptrBytes[i], 0xAB);
            }

            // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("invert block");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);
            for(int i = 0; i < ALLOC_SIZE/sizeof(uint32_t); i++)
            {
                ptr[i] = 0xABABABAB;
            }

                uint8_t* ptrBytes = (uint8_t*)ptr;

            uint32_t offset = 1;
            mem_status status = invert_block(ptr + offset, 4);

            for(int i = 0; i < ALLOC_SIZE; i++)
            {
                        UCUNIT_CheckIsEqual(ptrBytes[i], (i >= 4 && i < 8
) ? 0xAB^0xFF : 0xAB);
            }

            UCUNIT_CheckIsEqual(status, SUCCESS);
```

```
            // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        {
            UCUNIT_TestcaseBegin("invert block fail");
            uint32_t* ptr = allocate_words(ALLOC_SIZE);

            for(int i = 0; i < ALLOC_SIZE/sizeof(uint32_t); i++)
            {
                        ptr[i] = 0xABABABAB;
            }

                uint8_t* ptrBytes = (uint8_t*)ptr;

            uint32_t offset = 200;
            mem_status status = invert_block(ptr + offset, 4);

            for(int i = 0; i < ALLOC_SIZE; i++)
            {
                        UCUNIT_CheckIsEqual(ptrBytes[i], 0xAB);
            }

            UCUNIT_CheckIsEqual(status, FAILED);

            // clean up
            free_words(ptr);
            UCUNIT_TestcaseEnd();
        }

        return 0;
}
```

```
/*
 * @file main.c
 * @brief Project 3
 *
 * A set of functions for displaying and manipulating memory.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#include "memory_tests.h"
#include "handle_led.h"
#include "logger.h"
```

```
#include "setup_teardown.h"

/**
 * @brief The size of the allocation in this program.
 */
#define ALLOC_BYTES 16

int main()
{
        initialize();

        set_led(1, BLUE);

        log_string("\nAllocate 16 bytes (if necessary on your target
system, or specify the memory region in the heap)\n");
        uint32_t* ptr = allocate_words(ALLOC_BYTES);

        log_string("\nWrite a memory pattern to the allocated 16 byte
region using a selected seed\n");
        const uint8_t seed = 2;
        if(write_pattern(ptr, ALLOC_BYTES, seed) == FAILED)
                goto error;

        log_string("\nDisplay that region’s memory pattern\n");
        if(display_memory(ptr, ALLOC_BYTES) == NULL)
                goto error;

        log_string("\nVerify that region’s memory pattern (should
pass)\n");
        if(verify_pattern(ptr, ALLOC_BYTES, seed) != NULL)
                goto error;

        log_string("\nWrite 0xFFEE to a position within that region
(location + some offset you select)\n");
        const uint32_t offset = 4;
        uint32_t* write_location = get_address(ptr, offset);
        if(write_memory(write_location, 0xFF) == FAILED)
                goto error;

        write_location = get_address(ptr, offset+1);
        if(write_memory(write_location, 0xEE) == FAILED)
                goto error;

        log_string("\nDisplay that region’s memory pattern\n");

        uint32_t* read_location = get_address(ptr, offset);
        uint8_t* ptr_bytes = display_memory(read_location,
sizeof(uint16_t));
        if (!ptr_bytes || ptr_bytes[0] != 0xFF || ptr_bytes[1] != 0xEE)
                goto error;

        log_string("\nVerify the memory pattern again (should error)\n");
        if(verify_pattern(ptr, ALLOC_BYTES, seed) == NULL)
                goto error;
```

```
        log_string("\nWrite a memory pattern to the region using the same
seed as before\n");
        if(write_pattern(ptr, ALLOC_BYTES, seed) == FAILED)
                goto error;

        log_string("\nDisplay that region’s memory pattern\n");
        if(display_memory(ptr, ALLOC_BYTES) == NULL)
                goto error;

        log_string("\nVerify that memory pattern again (should pass)\n");
        if(verify_pattern(ptr, ALLOC_BYTES, seed) != NULL)
                goto error;

        log_string("\nInvert 4 bytes in the region (location + some
offset)\n");
        if(invert_block(get_address(ptr, offset), 4) == FAILED)
                goto error;

        log_string("\nDisplay that region’s memory pattern\n");
        if(display_memory(ptr, ALLOC_BYTES) == NULL)
                goto error;

        log_string("\nVerify the memory pattern again (should error)\n");
        if(verify_pattern(ptr, ALLOC_BYTES, seed) == NULL)
                goto error;

        log_string("\nInvert those 4 bytes again in the LMA region
(location + some offset)\n");
        if(invert_block(get_address(ptr, offset), 4) == FAILED)
                goto error;

        log_string("\nDisplay that region’s memory pattern\n");
        if(display_memory(ptr, ALLOC_BYTES) == NULL)
                goto error;

        log_string("\nVerify that memory pattern again (should pass)\n");
        if(verify_pattern(ptr, ALLOC_BYTES, seed) != NULL)
                goto error;

        log_string("\nFree the 16 byte allocated region (if necessary on
your target)\n");
        if(free_words(ptr) == FAILED)
                goto error;

        set_led(1, GREEN);
        terminate();
        return 0;

error:
    set_led(1, RED);
    terminate();
    return -1;
}
```

```
/*
 * @file handle_led.c
 * @brief Project 3
 *
 * Functions for handling the state of an LED.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */
#include "handle_led.h"
#include <stdio.h>

/**
 * set_led
 *
 * @brief Sets the LED state.
 * @details This function will simply print the
 *          state of what the LED would be.
 * @param inValue The on/off state of the LED to set.
 * @param inColor The color of the LED to set.
 */
void set_led(uint8_t inValue, enum COLOR inColor)
{
    printf("\nLED %s %s", COLOR_STRINGS[inColor], inValue ? "ON" : "OFF");
}
```

```
/*
 * @file logger.h
 * @brief Project 3
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef PES_PROJECT_3_LOGGER_H
#define PES_PROJECT_3_LOGGER_H
#include "logger.h"

#include <stdint.h>
```

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

/**
 * @brief Static variable that determines the state of the logger.
 */
static bool logging = false;

/**
 * @brief Log_enable - begin printing log messages when called
 */
void log_enable() {
        logging = true;
}

/**
 * @brief Log_disable - ignore any log messages until re-enabled
 */
void log_disable() {
        logging = false;
}

/**
 * @brief Log_status - returns a flag to indicate whether the logger is
enabled or disabled
 * @return Whether the log is currently enabled.
 */
bool log_enabled() {
        return logging;
}

/**
 * @brief Log_data - display in hexadecimal an address and contents of a
memory location,
 * @param inBytes a pointer to a sequence of bytes to log
 * @param inSize Number of bytes to log
 */
void log_data(const uint8_t* inBytes, size_t inSize) {
        if (logging) {
                printf("\nBytes at address
%p:\n=========================\n", inBytes);
                for(int i = 0; i < inSize; i++)
                {
                        printf("%2x ", inBytes[i]);
                        if((i+1)%4 == 0)
                        {
                                printf("\n");
                        }
                }
                printf("\n=========================\n");
        }
}
```

```
/**
 * @brief Display a string.
 * @param inString String to display.
 */
void log_string(const char* inString) {
        if (logging) {
                printf("%s\n", inString);
        }
}

/**
 * @brief Display an integer
 * @param inNum Integer to display.
 */
void log_integer(uint64_t inNum) {
        if (logging) {
                printf("%llu\n", inNum);
        }
}

#endif
```

```
/*
 * @file setup_teardown.h
 * @brief Project 3
 *
 * @details Contains the setup and cleanup prototypes.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#include "setup_teardown.h"
#include <stdint.h>
#include "logger.h"

/**
 * initialize
 *
 * @brief Print "program start" in debug builds. Shows that the program
successfully started.
 *
 */
void initialize(void)
{
#ifdef DEBUG
    log_enable();
```

```
#endif
}

/**
 * terminate
 *
 * @brief Print "program end" in debug builds. Shows that the program
successfully completed.
 *
 */
void terminate(void)
{
#ifdef DEBUG
    log_string("\nprogram end\n");
#endif
}
```

```
/*
 * @file setup_teardown.h
 * @brief Project 3
 *
 * @details Contains the setup and cleanup prototypes.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#include <stdio.h>
#include "board.h"
#include "peripherals.h"
#include "clock_config.h"
#include "MKL25Z4.h"
#include "fsl_debug_console.h"
#include "logger.h"
#include "fsl_gpio.h"
#include "pin_mux.h"


void initialize()
{
#ifdef DEBUG
        /* serial debug console setup: use PRINTF("debug msg"); */
        BOARD_InitDebugConsole();
        log_enable();
        PRINTF("\nprogram start\n");
#endif
```

```
    /* Board pin, clock, debug console init */
        BOARD_InitBootPins();
        BOARD_InitBootClocks();
        BOARD_InitBootPeripherals();

        /* led setup */
        LED_RED_INIT(1);
        LED_BLUE_INIT(1);
        LED_GREEN_INIT(1);

        LED_BLUE_OFF();
        LED_GREEN_OFF();
        LED_RED_OFF();
}

/**
 * terminate
 *
 * @details Print "program end" in debug builds.
 *          Shows that the program successfully completed.
 *
 */
void terminate()
{
#ifdef DEBUG
        PRINTF("\nprogram end\n");
#endif
}
```

```
/*
 * @file logger.h
 * @brief Project 3
 *
 * Tools for logging.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef PES_PROJECT_3_LOGGER_H
#define PES_PROJECT_3_LOGGER_H
#include "logger.h"

#include <stdint.h>
```

```
#include <stdbool.h>
#include <stdint.h>
#include "fsl_debug_console.h"

/**
 * @brief Static variable maintains the logging state.
 */
static bool logging = false;

/**
 * @brief Log_enable - begin printing log messages when called
 */
void log_enable() {
        logging = true;
}

/**
 * @brief Log_disable - ignore any log messages until re-enabled
 */
void log_disable() {
        logging = false;
}

/**
 * @brief Log_status - returns a flag to indicate whether the logger is
enabled or disabled
 * @return Whether the log is currently enabled.
 */
bool log_enabled() {
        return logging;
}

/**
 * @brief Log_data - display in hexadecimal an address and contents of a
memory location,
 * @param inBytes a pointer to a sequence of bytes to log
 * @param inSize Number of bytes to log
 */
void log_data(const uint8_t* inBytes, size_t inSize) {
        if (logging) {
                PRINTF("\nBytes at address
%p:\n=========================\n", inBytes);
                for(int i = 0; i < inSize; i++)
                {
                        PRINTF("%2x ", inBytes[i]);
                        if((i+1)%4 == 0)
                        {
                                PRINTF("\n");
                        }
                }
                printf("\n=========================\n");
        }
}
```

```
/**
 * @brief Display a string.
 * @param inString String to display.
 */
void log_string(const char* inString) {
        if (logging) {
                PRINTF("%s\n", inString);
        }
}

/**
 * @brief Display an integer
 * @param inNum Integer to display.
 */
void log_integer(uint64_t inNum) {
        if (logging) {
                PRINTF("%llu\n", inNum);
        }
}

#endif
```

```
/*
 * @file handle_led.c
 * @brief Project 3
 *
 * Functions for handling the state of an LED.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#include <stdint.h>
#include "handle_led.h"
#include "board.h"
#include "fsl_debug_console.h"
#include "fsl_gpio.h"
#include "MKL25Z4.h"

/**
 * set_led
 *
 * @brief Sets the LED state.
 * @details This function controls a physical LED and prints
 *          debug info over UART on debug builds.
 * @param inValue The on/off state of the LED to set.
 * @param inColor The color of the LED to set.
```

```
 */
void set_led(uint8_t inValue, enum COLOR inColor)
{
#ifdef DEBUG
        //GPIO_TogglePinsOutput(GPIOD, 1U << 7U);
        PRINTF("\nLED %s %s", COLOR_STRINGS[inColor], inValue ? "ON" :
"OFF");
#endif

        switch(inColor)
        {
                case RED:
                {
                        LED_BLUE_OFF();
                        LED_GREEN_OFF();
                        if(inValue)
                        {
                                LED_RED_ON();
                        }
                        else
                        {
                                LED_RED_OFF();
                        }

                        break;
                }
                case GREEN:
                {
                        LED_BLUE_OFF();
                        LED_RED_OFF();

                        if(inValue)
                        {
                                LED_GREEN_ON();
                        }
                        else
                        {
                                LED_GREEN_OFF();
                        }
                        break;
                }
                case BLUE:
                {
                        LED_GREEN_OFF();
                        LED_RED_OFF();

                        if(inValue)
                        {
                                LED_BLUE_ON();
                        }
                        else
                        {
                                LED_BLUE_OFF();
                        }
```

```
                        break;
                }
                default:
                        break;
        }
}
```

```
/*
 * @file memory_tests.c
 * @brief Project 3
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 *
 *          Leveraged code in this file includes the ABS macro, taken from
 *          Slide 30 of "More C Topics" lecture from ECEN 5813
 *          Link:
https://canvas.colorado.edu/courses/53078/files/folder/Class%20FIles?
preview=7085601
 */
#ifndef PES_PROJECT_3_MEM_TESTS_H
#define PES_PROJECT_3_MEM_TESTS_H

#include "memory_tests.h"

#include "mem_types.h"
#include "gen_pattern.h"
#include <stdlib.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

#include "logger.h"

/**
 * Length of the static arrays used for static memory alloc.
 */
#define ARRLEN 256

/**
 * ABS
 * @details Leveraged code in this file includes the ABS macro, taken from
 *          Slide 30 of "More C Topics" lecture from ECEN 5813
 *          Link:
```

```
https://canvas.colorado.edu/courses/53078/files/folder/Class%20FIles?
preview=7085601
 *  Takes a number and returns the absolute value of that number.
 */
#define ABS(x) ((x)>0?(x):-(x))

/**
 * Memory is tracked in this module using a linked
 * list of memory nodes.
 */
struct mem_list_node
{
        uint32_t* data;
        size_t size;
        struct mem_list_node* next;
};

/*
 * The head of the list is a statically allocated sentinel
 * used to simplify the linked list logic.
 */
static struct mem_list_node memListHead = { NULL, 0, NULL };

/**
 * Helper function to check whether an address resides within
 * memory that is currently allocated.
 */
bool addressIsOwned(uint32_t* inAddress)
{
        if(inAddress)
        {
                // try and see if the requested location resides in owned
memory
                struct mem_list_node* iter = &memListHead;
                while (iter)
                {
                        ptrdiff_t distance = ABS(iter->data - inAddress);
                        if(distance >= 0 &&
                           distance <= iter->size)
                        {
                                return true;
                        }
                        iter = iter->next;
                }
        }
        return false;
}

/**
 * Allocate a block of memory for the
 * test (using malloc()) - the argument for this
 * function is a number of bytes to allocate.
 * A maximum memory block size should be checked,
 * with an error thrown for an invalid allocation
```

```
 * request (execution can continue after the error) -
 * the routine should return a pointer to the allocated
 * memory.
 */
uint32_t * allocate_words(size_t length)
{
        if(length >= sizeof(uint32_t))
        {
                struct mem_list_node* iter = &memListHead;
                while(iter->next)
                {
                        iter = iter->next;
                }

                iter->next = (struct mem_list_node*)malloc(sizeof(struct
mem_list_node));
                iter->next->data = (uint32_t*)malloc(length);
                for(int i = 0; i < length; i++)
                {
                        ((uint8_t*)iter->next->data)[i] = 0x0;
                }

                iter->next->size = length;
                iter->next->next = NULL;

                return iter->next->data;
        }
        else
        {
                log_string("allocate_words: Trying to allocate invalid
memory ");
                log_integer(length);
                return NULL;
        }

}


/**
 *
 * Free a previously allocated block of
 * memory (using free()).  If free_memory
 * is called with no memory having been
 * allocated, a warning message should be
 * issued, but execution can continue.
 *
 */
mem_status free_words(uint32_t * src)
{
        if(src)
        {
                struct mem_list_node* iter = &memListHead;

                while (iter->next)
```

```
                {
                        if(iter->next->data == (void*)src)
                        {
                                free(src);
                                struct mem_list_node* node_to_delete =
iter->next;
                                iter->next = iter->next->next;
                                free(node_to_delete);
                                return SUCCESS;
                        }
                        iter = iter->next;
                }
        }
        return FAILED;
}

/**
 * contents of a memory region will be “displayed” by
 * returning the contents of memory at the location —
 * arguments to the function include a memory address
 * and a number of bytes to display.
 */
uint8_t * display_memory(uint32_t * loc, size_t length)
{
        static uint8_t arr[ARRLEN] = {0};
        for(int i = 0; i < ARRLEN; i++)
                arr[i] = 0;

        if(length <= ARRLEN)
        {
                for(int i = 0; i < length; i++)
                        arr[i] = *((uint8_t*)(loc) + i);

                log_data((uint8_t*)loc, length);

                return &arr[0];
        }
        return NULL;
}

/**
 * The user specifies an address and a byte value
 * to write.  The memory at that location is modified
 * accordingly.
 */
mem_status write_memory(uint32_t * loc, uint8_t value)
{
        if(addressIsOwned(loc))
        {
                memcpy(loc, &value, sizeof(uint8_t));
                return SUCCESS;
        }
        log_string("\nAddress [");
        log_data((uint8_t*)loc, sizeof(uint32_t*));
```

```
        log_string("] is not owned.");
        return FAILED;
}

/**
 * The user specifies an address and a number of bytes.
 * All bits in this region should be inverted using XOR.
 */
mem_status invert_block(uint32_t * loc, size_t length)
{
        if(addressIsOwned(loc))
        {
                uint8_t* locBytes = (uint8_t*)loc;

                log_string("\n Data Before invert: \n");
                log_data(locBytes, length);

                for(int i = 0; i < length; i++)
                {
                        uint8_t inverted_byte = locBytes[i];
                        inverted_byte ^= (uint8_t)0xFF;
                        locBytes[i] = inverted_byte;
                }

                log_string("\n Data After invert: \n");
                log_data(locBytes, length);
                return SUCCESS;
        }
        return FAILED;
}

/**
 * The user specifies an address, a number of bytes,
 * and a seed value.  The seed value and the number
 * of bytes will be provided to a pattern generator
 * utility function, which will return a byte array,
 * where each byte has a random value based on the seed.
 * The bytes returned will then be written into memory
 * starting at the specified address. (Note: the data type
 * for your seed value may vary.)
 */
mem_status write_pattern(uint32_t * loc, size_t length, uint8_t seed)
{
        if(addressIsOwned(loc) && length>0)
        {
                gen_pattern((uint8_t*)loc, length, seed);
                log_data((uint8_t*)loc, length);
                return SUCCESS;
        }
        return FAILED;
}

/**
 * The user specifies an address, a number of bytes, and
```

```
 * a seed value.  Using the seed value and the pattern
 * generator utility,  generate a byte array with random
 * values based on the seed.  Check whether the newly generated
 * pattern in the byte array returned matches the existing byte
 * pattern in memory at the specified address.  The function
 * should return a list of any memory location addresses
 * (compare bytes) where memory did not match the pattern.
 * (Note: the data type for your seed value may vary.)
 */
uint32_t * verify_pattern(uint32_t * loc, size_t length, uint8_t seed)
{
        static uint8_t arr[ARRLEN] = {0};
        for(int i = 0; i < ARRLEN; i++)
                arr[i] = 0;

        uint32_t nextNonMatch = 0;
        bool matched = true;
        if(addressIsOwned(loc))
        {
                uint32_t* cmp_pattern = allocate_words(length);
                gen_pattern((uint8_t*)cmp_pattern, length, seed);
                uint8_t* first_pattern_bytes = (uint8_t*)loc;
                uint8_t* cmp_pattern_bytes = (uint8_t*)cmp_pattern;

                log_string("Input data:\n");
                log_data((uint8_t*)loc, length);

                log_string("Compared data:\n");
                log_data(cmp_pattern_bytes, length);

                for(int i = 0; i < length; i++)
                {
                        if(first_pattern_bytes[i] != cmp_pattern_bytes[i])
                        {
                                arr[nextNonMatch++] =
first_pattern_bytes[i];
                                matched = false;
                        }
                }
                free_words(cmp_pattern);
        }

        return matched ? NULL : (uint32_t*)&arr[0];
}

/**
 * this function will take an offset from a specified
 * location and calculate a fully addressed memory location
 * for use in the functions above.  In this way, memory can
 * be more easily addressed by an offset in the region intended
 * for use.
 */
uint32_t * get_address(uint32_t * base, uint32_t offset)
{
```

```
        if(base)
        {
                return (uint32_t*)(((uint8_t*)base)+offset);
        }
        return NULL;
}



#endif
```

```
/*
 * @file gen_pattern.h
 * @brief Project 3
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 *
 *
 */

#ifndef GENPATTERNH
#define GENPATTERNH

#include <stdint.h>
#include <stddef.h>

// http://codeforces.com/blog/entry/61587
static long holdrand = 1L;

void SRAND(unsigned int seed) {
    holdrand = (long) seed;
}

long RAND() {
    return (((holdrand = holdrand * 214013L + 2531011L) >> 16) & 0x7fff);
}

//pattern generation function will accept a number of bytes and a seed
value and return a byte array.
void gen_pattern(uint8_t * pattern, size_t length, uint8_t seed)
{
        if(pattern)
        {
                SRAND(seed);
```

```
                for(int i = 0; i < length; i++)
                {
                        pattern[i] = (uint8_t)RAND();
                }
        }
}

#endif
```

```
/*
 * @file gen_pattern.h
 * @brief Project 3
 *
 * Functions for generating a random pattern.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef GENPATTERNH
#define GENPATTERNH

#include <stdint.h>
#include <stddef.h>

/**
 * @brief Pattern generation function will accept a number of bytes and a
seed value and return a byte array.
 * @param outPattern The region of memory to write the pattern to.
 * @param inLength The number of bytes to make the pattern.
 * @param inSeed The seed to use in pattern generation.
 */
void gen_pattern(uint8_t * outPattern, size_t inLength, uint8_t inSeed);


#endif
```

```
/*
 * @file handle_led.h
 * @brief Project 3
 *
 * @details Contains the prototype for handling LEDs on various platforms.
```

```
 *          This may be actually turning an LED on and off or just
printing
 *          what the LED state would be, in the absence of LEDs.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */
#ifndef PES_PROJECT_2_HANDLE_LED_H
#define PES_PROJECT_2_HANDLE_LED_H

#include <stdint.h>
#include "led_types.h"

/**
 * set_led
 *
 * @brief Sets the LED state.
 * @details This function, depending on platform, may or may not
 *          control a physical LED. On PC, it will simply print the
 *          state of what the LED would be.
 * @param inValue The on/off state of the LED to set.
 * @param inColor The color of the LED to set.
 */
void set_led(uint8_t inValue, enum COLOR inColor);

#endif //PES_PROJECT_2_HANDLE_LED_H
```

```
/*
 * @file led_types.h
 * @brief Project 3
 *
 * @details Defines enumerations and constants used to describe colors and
 *          on/off states for LEDs.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef PES_PROJECT_2_LED_TYPES_H
#define PES_PROJECT_2_LED_TYPES_H
```

```
/**
 * COLOR
 *
 * @brief The possible color values of the LED.
 */
enum COLOR
{
    RED = 0,
    GREEN,
    BLUE,
    NUM_COLORS
};

/**
 * COLOR_STRINGS
 *
 * @brief String representations of the COLOR enum, used for printing.
 */
static const char * const COLOR_STRINGS[3] = {
        "RED",
        "GREEN",
        "BLUE"
};

#endif //PES_PROJECT_2_LED_TYPES_H
```

```
/*
 * @file logger.h
 * @brief Project 3
 *
 * Interface to use for logging on either PC or KL25Z.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef PES_PROJECT_3_LOGGER_H
#define PES_PROJECT_3_LOGGER_H

#include <stdbool.h>
#include <stddef.h>


/**
```

```
 * @brief Log_enable - begin printing log messages when called
 */
void log_enable();

/**
 * @brief Log_disable - ignore any log messages until re-enabled
 */
void log_disable();

/**
 * @brief Log_status - returns a flag to indicate whether the logger is
enabled or disabled
 * @return Whether the log is currently enabled.
 */
bool log_enabled();


/**
 * @brief Log_data - display in hexadecimal an address and contents of a
memory location,
 * @param inBytes a pointer to a sequence of bytes to log
 * @param inSize Number of bytes to log
 */
void log_data(const uint8_t* inBytes, size_t inSize);

/**
 * @brief Display a string.
 * @param inString String to display.
 */
void log_string(const char* inString);

/**
 * @brief Display an integer
 * @param inNum Integer to display.
 */
void log_integer(uint64_t inNum);

#endif
```

```
/*
 * @file mem_types.h
 * @brief Project 3
 *
 * Defines to be used by memory functions.
 *
 * @author Jack Campbell
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
```

```
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef INCLUDE_MEM_TYPES_H_
#define INCLUDE_MEM_TYPES_H_

/**
 * @brief Return code for memory functions.
 */
typedef enum mem_status
{
        SUCCESS = 0,
        FAILED
} mem_status;


#endif /* INCLUDE_MEM_TYPES_H_ */
```

```
/*
 * @file memory_tests.h
 * @brief Project 3
 *
 * A series of functions for manipulating and inspecting memory.
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */
#ifndef PES_PROJECT_3_MEM_TESTS_H
#define PES_PROJECT_3_MEM_TESTS_H

#include <stdint.h>
#include <stddef.h>
#include "mem_types.h"

/**
 * @brief Allocate a block of memory for the
 * test (using malloc()) - the argument for this
 * function is a number of bytes to allocate.
 * A maximum memory block size should be checked,
 * with an error thrown for an invalid allocation
 * request (execution can continue after the error) -
 * the routine should return a pointer to the allocated
 * memory.
 * @param length Number of bytes to allocate.
 * @return Pointer to allocated memory.
```

```
 */
uint32_t * allocate_words(size_t length);

/**
 * @brief Free a previously allocated block of
 * memory (using free()).  If free_memory
 * is called with no memory having been
 * allocated, a warning message should be
 * issued, but execution can continue.
 * @param src Pointer to the memory to free.
 * @return Status of the free. SUCCESS unless
 * the src pointer is invalid.
 */
mem_status free_words(uint32_t * src);

/**
 * @brief contents of a memory region will be “displayed” by
 * returning the contents of memory at the location —
 * arguments to the function include a memory address
 * and a number of bytes to display.
 * @param loc Address of the memory to display
 * @param length Number of bytes to diplay
 * @return Byte array that is length long.
 */
uint8_t * display_memory(uint32_t * loc, size_t length);

/**
 * @brief The user specifies an address and a byte value
 * to write.  The memory at that location is modified
 * accordingly.
 * @param loc Address to write to
 * @param value Value to write at the address.
 * @return Whether the operation succeeded.
 */
mem_status write_memory(uint32_t * loc, uint8_t value);


/**
 * @brief The user specifies an address and a number of bytes.
 * All bits in this region should be inverted using XOR.
 * @param loc Address to start the invert operation
 * @param length How many bytes to invert
 * @return Whether the operation succeeded.
 */
mem_status invert_block(uint32_t * loc, size_t length);


/**
 * @brief The user specifies an address, a number of bytes,
 * and a seed value.  The seed value and the number
 * of bytes will be provided to a pattern generator
 * utility function, which will return a byte array,
 * where each byte has a random value based on the seed.
 * The bytes returned will then be written into memory
```

```
 * starting at the specified address. (Note: the data type
 * for your seed value may vary.)
 * @param loc The address to write the pattern to.
 * @param length How many bytes the pattern should be.
 * @param seed Seed to use in random number generator.
 * @return Whether the operation succeeded.
 */
mem_status write_pattern(uint32_t * loc, size_t length, uint8_t seed);


/**
 * @brief The user specifies an address, a number of bytes, and
 * a seed value.  Using the seed value and the pattern
 * generator utility,  generate a byte array with random
 * values based on the seed.  Check whether the newly generated
 * pattern in the byte array returned matches the existing byte
 * pattern in memory at the specified address.  The function
 * should return a list of any memory location addresses
 * (compare bytes) where memory did not match the pattern.
 * (Note: the data type for your seed value may vary.)
 * @param loc The address of the region to verify against
 * @param length How many bytes long is the region
 * @param seed The seed used in pattern generation
 * @return A list of any bytes that did not match.
 */
uint32_t * verify_pattern(uint32_t * loc, size_t length, uint8_t seed);


/**
 * @brief this function will take an offset from a specified
 * location and calculate a fully addressed memory location
 * for use in the functions above.  In this way, memory can
 * be more easily addressed by an offset in the region intended
 * for use.
 * @param base The base of the address to get
 * @param offset How many bytes to offset the address.
 * @return The offset address. May not be aligned.
 */
uint32_t * get_address(uint32_t * base, uint32_t offset);



#endif //PES_PROJECT_2_DELAY_H
```

```
/*
 * @file setup_teardown.h
 * @brief Project 3
 *
 * @details Contains the setup and cleanup prototypes to be implemented
 *          both for the FB and PC variants of the build.
```

```
 *
 * @tools  PC Compiler: GNU gcc 8.3.0
 *         PC Linker: GNU ld 2.32
 *         PC Debugger: GNU gdb 8.2.91.20190405-git
 *         ARM Compiler: GNU gcc version 8.2.1 20181213
 *         ARM Linker: GNU ld 2.31.51.20181213
 *         ARM Debugger: GNU gdb 8.2.50.20181213-git
 */

#ifndef PES_PROJECT_2_SETUP_TEARDOWN_H
#define PES_PROJECT_2_SETUP_TEARDOWN_H

/**
 * initialize
 *
 * @details Initializes components needed by a particular platform,
 *          such as LEDs.
 *
 */
void initialize(void);


/**
 * terminate
 *
 * @details Cleans up any required components on a particular platform.
 *
 */
void terminate(void);

#endif //PES_PROJECT_2_SETUP_TEARDOWN_H
```

```
# defines for the pc builds
PC_SRC_REL_ROOT = ../source/pc_implementation
COMMON_REL_ROOT = ../source/common
PC_SRCS = $(wildcard $(PC_SRC_REL_ROOT)/*.c $(COMMON_REL_ROOT)/*.c)
PC_OBJS = $(patsubst ../%.c, bin/%.o, $(PC_SRCS))
# compiler
CC = gcc
# flags
CFLAGS  = -g -Wall -Werror -I../include  -I../uCUnit -DDEBUG -I../board -
I../include -I../../include -I../source -I../ -I../drivers -I../CMSIS -
I../utilities -I../startup
RM := rm -rf

# runs all rules
all: pc_debug

# dependencies create the output and build dirs and then compile/link all
the code
```

```
pc_debug: output/pc_debug

# Create the output dir and place
output/pc_debug: $(PC_OBJS) bin/source/pc_implementation/main.o
	mkdir -p $(dir $@)
	$(CC) $(CFLAGS) -DDEBUG -o $@ $^

# making object targets for all the source files
bin/source/pc_implementation/%.o: $(PC_SRC_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS) -DDEBUG

bin/source/common/%.o: $(COMMON_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS) -DDEBUG

# special case main, which is not in the platform specific dir
bin/source/pc_implementation/main.o: ../source/main.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS) -DDEBUG

# Other Targets
clean:
	-$(RM) $(EXECUTABLES)$(OBJS)$(C_DEPS)$(PC_OBJS) output/* bin/*
output/pes_project3.axf source/pc_implementation/main.o */*.o */*/*.o
*/*.su */*/*.su */*.d */*/*.d
	-@echo ' '
```

```
# defines for the pc builds
PC_SRC_REL_ROOT = ../source/pc_implementation
COMMON_REL_ROOT = ../source/common
PC_SRCS = $(wildcard $(PC_SRC_REL_ROOT)/*.c $(COMMON_REL_ROOT)/*.c)
PC_OBJS = $(patsubst ../%.c, bin/%.o, $(PC_SRCS))
# compiler
CC = gcc
# flags
CFLAGS  = -g -Wall -Werror -I../include  -I../uCUnit -DDEBUG -I../board -
I../include -I../../include -I../source -I../ -I../drivers -I../CMSIS -
I../utilities -I../startup
RM := rm -rf

# runs all rules
all: pc_debug_test

# dependencies create the output and build dirs and then compile/link all
the code

pc_debug_test: output/pc_debug_test

output/pc_debug_test: $(PC_OBJS) bin/tests/main.o bin/uCUnit/pc/System.o
	mkdir -p $(dir $@)
```

```
        $(CC) $(CFLAGS) -o $@ $^

# making object targets for all the source files
bin/source/pc_implementation/%.o: $(PC_SRC_REL_ROOT)/%.c
        mkdir -p $(dir $@)
        gcc -c -o $@ $< $(CFLAGS) -DDEBUG

bin/source/common/%.o: $(COMMON_REL_ROOT)/%.c
        mkdir -p $(dir $@)
        gcc -c -o $@ $< $(CFLAGS) -DDEBUG

# special case main, which is not in the platform specific dir
bin/tests/main.o: ../tests/main.c
        echo "COMPILING TEST MAIN"
        mkdir -p $(dir $@)
        gcc -c -o $@ $< $(CFLAGS)

bin/uCUnit/pc/System.o: ../uCUnit/pc/System.c
        mkdir -p $(dir $@)
        gcc -c -o $@ $< $(CFLAGS)

# Other Targets
clean:
        -$(RM) $(EXECUTABLES)$(OBJS)$(C_DEPS)$(PC_OBJS) output/pc_debug
output/pes_project3.axf source/pc_implementation/main.o */*.o */*/*.o
*/*.su */*/*.su */*.d */*/*.d
        -@echo ' '

post-build:
        -@echo 'Performing post-build steps'
        -arm-none-eabi-size "output/pes_project3.axf"; # arm-none-eabi-
objcopy -v -O binary "pes_project3.axf" "pes_project3.bin" ; # checksum -p
MKL25Z128xxx4 -d "pes_project3.bin";
        -@echo ' '

.PHONY: all clean dependents post-build
```

```
# defines for the pc builds
PC_SRC_REL_ROOT = ../source/pc_implementation
COMMON_REL_ROOT = ../source/common
PC_SRCS = $(wildcard $(PC_SRC_REL_ROOT)/*.c $(COMMON_REL_ROOT)/*.c)
PC_OBJS = $(patsubst ../%.c, bin/%.o, $(PC_SRCS))
# compiler
CC = gcc
# flags
CFLAGS  = -g -Werror -Wall -I../include -I../uCUnit
RM := rm -rf

# dependencies create the output and build dirs and then compile/link all
the code
```

```
pc_run: output/pc_run

output/pc_run: $(PC_OBJS)  bin/source/pc_implementation/main.o
	echo "PC OBJS:" $(PC_OBJS)
	mkdir -p $(dir $@)
	$(CC) $(CFLAGS) -o $@ $^


# making object targets for all the source files
bin/source/pc_implementation/%.o: $(PC_SRC_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

bin/source/common/%.o: $(COMMON_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

# special case main, which is not in the platform specific dir
bin/source/pc_implementation/main.o: ../source/main.c
	echo "COMPILING APP MAIN"
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

# Other Targets
clean:
	-$(RM) $(EXECUTABLES)$(OBJS)$(C_DEPS) bin/* output/* */*.o */*/*.o
*/*.su */*/*.su */*.d */*/*.d
	-@echo ' '
```

```
# defines for the pc builds
PC_SRC_REL_ROOT = ../source/pc_implementation
COMMON_REL_ROOT = ../source/common
PC_SRCS = $(wildcard $(PC_SRC_REL_ROOT)/*.c $(COMMON_REL_ROOT)/*.c)
PC_OBJS = $(patsubst ../%.c, bin/%.o, $(PC_SRCS))
# compiler
CC = gcc
# flags
CFLAGS  = -g -Werror -Wall -I../include -I../uCUnit
RM := rm -rf

# runs all rules
all: pc_run_test

# dependencies create the output and build dirs and then compile/link all
the code
pc_run_test: output/pc_run_test

# Create the main executable
output/pc_run_test: $(PC_OBJS) bin/tests/main.o bin/uCUnit/pc/System.o
```

```
	mkdir -p $(dir $@)
	$(CC) $(CFLAGS) -o $@ $^

# making object targets for all the source files
bin/source/pc_implementation/%.o: $(PC_SRC_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

bin/source/common/%.o: $(COMMON_REL_ROOT)/%.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

# Also compile in the uCUnit test files
bin/tests/main.o: ../tests/main.c
	echo "COMPILING TEST MAIN"
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

bin/uCUnit/pc/System.o: ../uCUnit/pc/System.c
	mkdir -p $(dir $@)
	gcc -c -o $@ $< $(CFLAGS)

# Other Targets
clean:
	-$(RM) $(EXECUTABLES)$(OBJS)$(C_DEPS) output/* bin/* */*.o */*/*.o
*/*.su */*/*.su */*.d */*/*.d
	-@echo ' '
```

```
# Top level makefile. Delves into the Release and Debug subdirs and runs
make on the specific targets contained.

# Builds all targets.
all: fb_run fb_debug fb_run_test fb_debug_test

# Builds the PC release build.
pc_run:
	cd PC_Release && $(MAKE) pc_run

pc_debug:
	cd PC_Debug && $(MAKE) pc_debug

# Builds the PC release build.
pc_run_test:
	cd PC_Release_Test && $(MAKE) pc_run_test

pc_debug_test:
	cd PC_Debug_Test && $(MAKE) pc_debug_test

# Builds the FB release build.
```

```
fb_run:
        cd KL25Z_Release && $(MAKE) output/kl25z_run.axf

fb_debug:
        cd KL25Z_Debug && $(MAKE) output/kl25z_debug.axf

fb_run_test:
        cd KL25Z_Release_Test && $(MAKE) output/kl25z_run_test.axf

fb_debug_test:
        cd KL25Z_Debug_test && $(MAKE) output/kl25z_debug_test.axf

# Cleans both Debug and Release areas.
clean:
        cd KL25Z_Debug && $(MAKE) clean
        cd PC_Debug && $(MAKE) clean
        cd KL25Z_Release && $(MAKE) clean
        cd PC_Release && $(MAKE) clean
        cd KL25Z_Debug_test && $(MAKE) clean
        cd PC_Debug_Test && $(MAKE) clean
        cd KL25Z_Release_Test && $(MAKE) clean
        cd PC_Release_Test && $(MAKE) clean

.PHONY: pc_debug pc_run pc_run_test pc_debug_test
```

```
# Top level makefile. Delves into the Release and Debug subdirs and runs
make on the specific targets contained.

# Builds all targets.
all: pc_run pc_debug pc_run_test pc_debug_test

# Builds the PC release build.
pc_run:
        cd PC_Release && $(MAKE) pc_run

pc_debug:
        cd PC_Debug && $(MAKE) pc_debug

# Builds the PC release build.
pc_run_test:
        cd PC_Release_Test && $(MAKE) pc_run_test

pc_debug_test:
        cd PC_Debug_Test && $(MAKE) pc_debug_test

# Builds the FB release build.
fb_run:
        cd KL25Z_Release && $(MAKE) output/kl25z_run.axf

fb_debug:
        cd KL25Z_Debug && $(MAKE) output/kl25z_debug.axf
```

```
fb_run_test:
        cd KL25Z_Release_Test && $(MAKE) output/kl25z_run_test.axf

fb_debug_test:
        cd KL25Z_Debug_test && $(MAKE) output/kl25z_debug_test.axf

# Cleans both Debug and Release areas.
clean:
        cd KL25Z_Debug && $(MAKE) clean
        cd PC_Debug && $(MAKE) clean
        cd KL25Z_Release && $(MAKE) clean
        cd PC_Release && $(MAKE) clean
        cd KL25Z_Debug_test && $(MAKE) clean
        cd PC_Debug_Test && $(MAKE) clean
        cd KL25Z_Release_Test && $(MAKE) clean
        cd PC_Release_Test && $(MAKE) clean

.PHONY: pc_debug pc_run pc_run_test pc_debug_test
```