Andrew Courtemanche
Fri, Dec. 18th, 2020
ECE 649

Final Project Report

Section 1: Motivation

Microcontrollers such as the MSP430 are strictly single core, meaning they can only have a single logical thread of execution. This limitation complicates software design when a microcontroller needs to handle several tasks simultaneously. Tasks that are logically independent cannot be separated because of the single execution loop. Great care must be taken to ensure time-sensitive code runs exactly when required. Low power modes must be explicitly entered and handled, while ensuring the core will wake when required.

A Real-Time Operating System (henceforth RTOS) solves this problem. Multiple processes can run independently with their own stacks and contexts. Strict scheduling rules ensure time-sensitive tasks run exactly when they need to. Pre-Emptive scheduling allows processes to run with an illusion of parallelism, simplifying software design. The OS can automatically enter and exit low power states without processes needing any explicit knowledge of the processor state. Use of a RTOS allows for faster, less error prone development with a minor cost to performance and memory usage.

The Texas Instruments MSP430 family is a series of 16-bit microcontrollers optimized for low power applications. They are not designed for high performance and have very limited memory (2KB SRAM maximum across all models). They do not have hardware memory management units (MMU) which are required for any form of virtual memory addressing, so once a process is given memory it cannot be relocated. Since each running process requires its own stack, high process counts would be prohibitively memory intensive.

The MSP430FRxxxx family provides a solution. They contain up to 256KB of high-performance non-volatile Ferro-electric RAM. It is an order of magnitude faster than traditional flash memory, has practically unlimited write endurance, and can write at up to 8MHz. These properties make moving process stacks to FRAM possible with minimal performance loss.

The MSP430FR6989 contains 128KB of memory-mapped FRAM. ~48KB is mapped to 16-bit address space, while a further ~82KB is mapped to 20-bit address space. Accessing 20-bit address space requires extended machine instructions, requiring an extra word fetch while decoding the instruction. This means it is preferable to put the process stacks into the lower 48KB of the FRAM if possible. The upper 82KB of FRAM is ideal for code, as CPU prefetch minimizes the overhead of this slower access. The FRAM controller also has a built-in read cache for further overhead reduction.

This project report details the implementation of a basic RTOS, henceforth referred to as ARCOS (simply an acronym of the author's name and "Operating System"). ARCOS uses a portion of the lower 48KB of FRAM to give each process a dedicated stack for execution. A periodic watchdog timer interrupt saves the state of the current process and suspends execution, then control flow returns to the OS. The OS takes care of any “housekeeping” tasks it needs to, then selects the next process to be executed. That process’s context is restored, and execution continues. This means that the CPU would run a single process for a certain amount of time (a “timeslice”) before switching to another available process. This gives the illusion of concurrency.

Section 2: Implementation Details and Design Diagram

Given the unconventional nature of the code in this project, a detailed understanding of the MSP430 processor’s Instruction Set Architecture (ISA) and Application Binary Interface (ABI), hardware features, and compiler behavior was necessary. The MSP430 does not have any dedicated hardware to facilitate a RTOS, so manual manipulation of assembly instructions, ABI, stack and stack pointers, linker blob placement, and other less-than-ideal methods were required.

A program using ARCOS first calls arcos_init(), which configures registers, sets clocks, initializes any memory, and anything else that needs to be done ahead of time. The program is then free to run any of its own initialization code. The last call in the main() function should be a call to arcos_start(), which hands execution over to ARCOS. At this point, the current execution stack is invalidated as ARCOS moves the stack pointer to its own internal pre-allocated stack. This means that arcos_start() will never return, and nothing that was allocated on the original stack should be accessed again. arcos_start() will then transfer execution to arcos_os_run(). arcos_os_run() will always run using the pre-allocated stack for the OS. arcos_os_run() will then call arcos_os_schedule(). This function selects a process from the list of currently active processes in a priority-based round-robin fashion. This means that all tasks of a given priority will be chosen in a round-robin fashion, and processes will never be chosen if there is a process with higher priority ready. This is not an ideal scheduling system, but it is simple and fast.

Once a process is selected, that process’s stack pointer is restored. Then the entire execution context of that process is restored by popping all general-purpose CPU registers off the stack. Finally, the program counter is restored to where the process was interrupted. The process of carefully restoring the process’s entire context without error is impossible to do in plain C, use of assembly instructions is required. Furthermore, compiler directives must be set to make sure the compiler does not interfere with restoration (this is detailed in *Section 5*).

At this point, the chosen process can now continue execution where it left off. The program is entirely unaware that it was interrupted. Execution will continue until one of three things occurs:

1. The process returns
2. The process calls arcos_proc_terminate() on itself
3. The Watchdog Timer Interval interrupt occurs

If the process returns or calls arcos_proc_terminate() on itself, the OS removes that process from its internal process list and frees associated memory, if applicable. The process will not run again unless recreated and restarted. If the Watchdog Timer Interval interrupt occurs, the process is “pre-empted”. The Watchdog Timer Interval vector calls arcos_os_isr_timeout_slice() which very carefully saves the context of the current process to the stack. The order in which the context is saved to the stack is critical so that the process can be correctly restored later. Again, this requires careful assembly and compiler directives to ensure proper behavior. This cycle of Schedule->Execute->Pre-empt continues indefinitely.

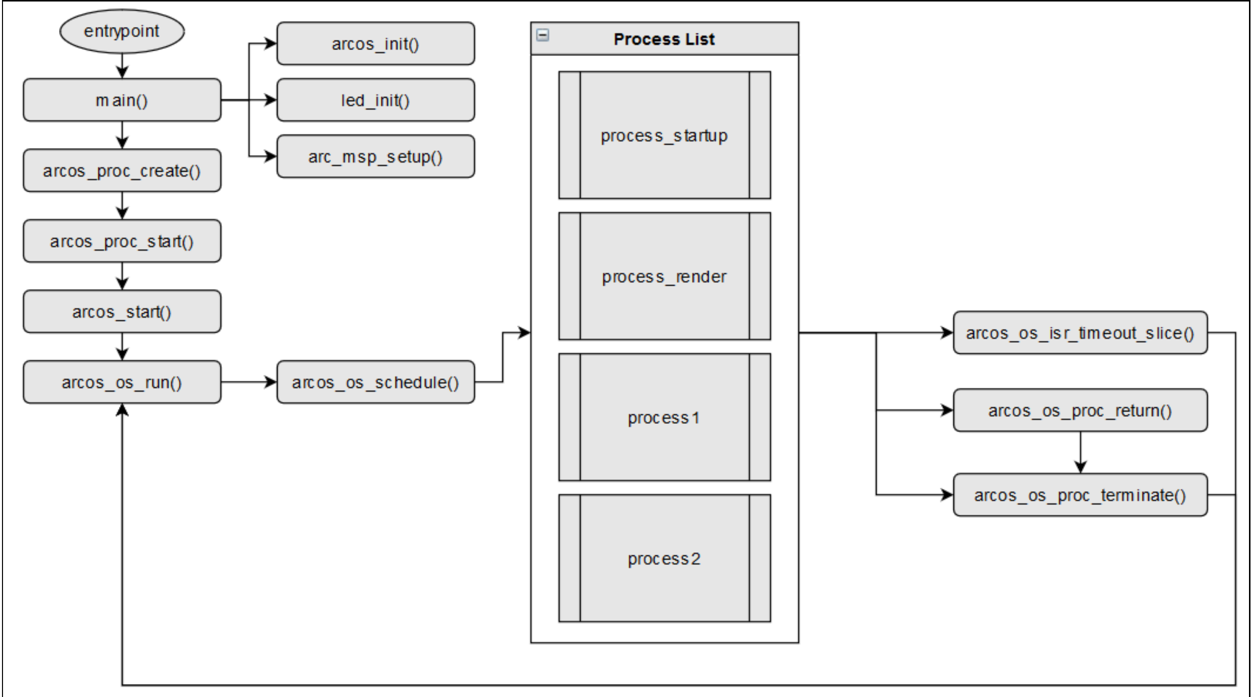
To demonstrate the system’s ability to manage multiple tasks simultaneously, an example program was written that has multiple independent tasks. These tasks are intentionally unrelated from each other to show they are operating independently. In this case, there are three tasks:

1. Illuminate the Red LED when the left button is pressed
2. Illuminate the Green LED when the right button is pressed
3. Constantly draw a 32x32 RGB framebuffer and output it to an external LED panel

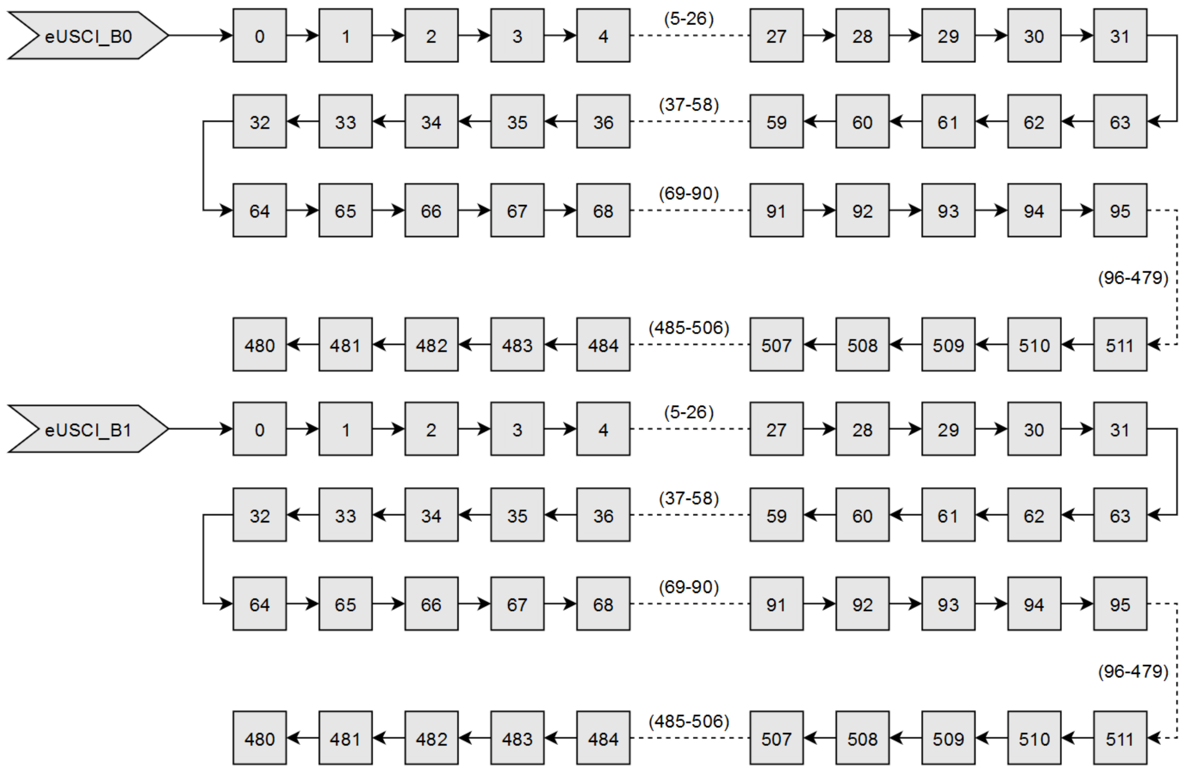
There is a separate task for each LED/Button pair. They are completely independent on one another, and each of them is implemented as a simple infinite “while(true)” loop.

The 32x32 panel is split into two banks of 32 LEDs wide and 16 LEDs high. It is constructed out of two strips of WS2812B individually addressable RGB LEDs. The strips are wired in an “S” pattern, meaning every other row is shifted in backwards, which must be corrected in software. The example program fills a 32*32*3 RGB framebuffer with an alternating gradient test pattern every other frame. This forces the CPU to do a significant amount of work at regular intervals. This framebuffer is then output to the LED panel for display once the frame is complete.

Design Diagram



LED Panel Construction



*This illustration shows the construction of the 32x32 (WxH) WS2812B LED panel. It shows the two 32x16 halves of the panel and how they are wired in alternating directions row-by-row. It is important to note that the indexes are LED indexes and are not in the same order as data is shifted in. Therefore, every other row must be reversed before shifting it out to the panel. The panel is split into two halves to increase the speed at which the panel can be refreshed.

Section 3: Parameter Selection

ARCOS Configuration:

Watchdog Timer	
WDTCTL	`WDTPW | WDTHOLD | WDTSSEL__ACLK | WDTTMSEL | WDTCNTCL | WDTIS__64;`
	ACLK source (32,768Hz), Interval Timer mode, Clear counter, /64 clock divider
SFRIE1	`WDTIE;`
	Watchdog Timer Interrupt Enable

*WDTHOLD is only enabled temporarily

Clock System	
CSCTL1	`DCORSEL | DCOFSEL_4;`
	DCO high frequency range, DCO set to 16MHz
CSCTL2	`SELA__LFXTCLK | SELS__MODCLK | SELM__DCOCLK;`
	LFXTCLK as ACLK source, MODCLK as SMCLK source, DCOCLK as MCLK source
CSCTL3	`DIVA__1 | DIVS__2 | DIVM__1;`
	Divide ACLK source by 1, Divide SMCLK source by 2, Divide MCLK source by 1

*This configuration results in 32,768Hz for ACLK, 2.5MHz for SMCLK and 16MHz for MCLK

LED Panel Configuration:

eUSCI_B (Channels 0 and 1)	
UCBxBRW	`0;`
	Do not divide BRCLK
UCBxCTLW0	`UCMSB | UCMST | UCSYNC | UCSSEL__SMCLK;`
	MSB first, Master mode, SPI, SMCLK as BRCLK source

*This configuration results in a 2.5MHz bit clock (0.4 µs period)

DMA (Channels 0 and 1)	
DMACTL0	`DMA0TSEL__UCB0TXIFG0 | DMA1TSEL__UCB1TXIFG0;`
	Use UCBxTXIFG0 as DMA trigger
DMAxCTL	`DMADT_0 | DMASRCINCR_3 | DMADSTBYTE | DMASRCBYTE;`
	Single transfer mode, increment source address, byte destination, byte source
DMA0DA	`0x0640 + 0x000E;`
	UCB0TXBUF as destination address
DMA1DA	`0x0680 + 0x000E;`
	UCB1TXBUF as destination address
DMAxSZ	`9;`
	Transfer 9 bytes

GPIO P1.6 and P4.0	
P1SEL0_6	`1;`
	Select GPIO pin function 1 (UCB0SIMO)
P1SEL1_6	`0;`
	Select GPIO pin function 1 (UCB0SIMO)
P4SEL0_0	`0;`
	Select GPIO pin function 2 (UCB1SIMO)
P4SEL1_0	`1;`
	Select GPIO pin function 2 (UCB1SIMO)

GPIO Configuration:

Green LED GPIO P9.7	
P9DIR_7	`1;`
	Output
P9OUT_7	`0;`
	Initially LOW (LED OFF)

Red LED GPIO P1.0	
P1DIR_0	`1;`
	Output
P1OUT_0	`0;`
	Initially LOW (LED OFF)

Left Pushbutton GPIO P1.1	
P1DIR_1	`0;`
	Input
P1OUT_1	`1;`
	Pull-up resistor
P1REN_1	`1;`
	Enable pull up/down resistor

Right Pushbutton GPIO P1.2	
P1DIR_2	`0;`
	Input
P1OUT_2	`1;`
	Pull-up resistor
P1REN_2	`1;`
	Enable pull up/down resistor

Power Management	
PM5CTL0	`&= ~LOCKLPM5;`
	Unlock GPIO pins

Section 4: Demonstration

https://andrewcourtemanche.com/videos/ECE649_FinalProject_Video_Andrew_Courtemanche.mp4

This video demonstrates the multitasking capabilities of ARCOS by showing that the MCU is simultaneously running three separate processes:

-Checking the left button and turning the red LED on if it is pressed
-Checking the right button and turning the green LED if it is pressed
-Drawing and displaying a new framebuffer at regular intervals

None of these processes must be written in a way that gives control flow away, they are all simple infinite while(true) loops. Normally, such a loop would mean only the code within the loop can run. The multitasking of ARCOS allows each process to be written in a simple and intuitive way, isolating logically independent tasks. That is to say each process can be treated as its own main() function.

Section 5: Debug History

The entirety of the development of this project was done within TI's Code Composer Studio (CCS). CCS has an *excellent* set of debugging tools, which made debugging the complicated non-standard behavior of ARCOS a lot less painful. Regardless, most of the development time was spent combing through memory, assembly instructions, and registers manually. Here I will attempt to highlight some of the more interesting issues encountered during development.

Toolchain issues:

By default, CCS uses the included TI compiler. For most use cases, it is very capable and optimized to make best use of MSP430 features/restrictions. However, I quickly ran into problems. ARCOS requires several compiler features that the TI compiler simply does not support. Most critically, the TI compiler does not have a way to designate a function as "naked", as in, no prologue or epilogue where registers are pushed to the stack. For this reason alone, switching to a different compiler was required. Luckily, CCS has built in support for GCC, the GNU Compiler Collection, which *does* support the features needed for ARCOS.

The switch was not seamless, however. GCC does not have the same first-class support in CCS as the TI compiler, so a lot of the toolchain needed to be configured manually. There were issues with spaces in path names, a different set of compiler/linker flags, and other strange issues that needed to be worked out. Furthermore, GCC does not support the same pragma directives as the TI compiler, so those needed to be manually changed over. This was most notably a problem with Interrupt Service Routine vector pragmas, which needed to be switched over to GCC's attributes. For example, `#pragma vector = PORT1_VECTOR` would need to be replaced with `__attribute__ ((interrupt(PORT1_VECTOR)))`. Other tools do not work with GCC, such as the "Stack Usage" tool.

Since ARCOS makes a lot of changes outside the scope of the compiler, a lot of care had to be made to make sure the compiler would not make assumptions that are no longer valid, especially with optimizations enabled. Here are just some of the things that need to be carefully handled:

-Function addresses for each process are passed to ARCOS to be run manually but are never called within the compilation unit. Therefore, the compiler would remove the seemingly unused functions, causing a lockup when ARCOS went to call it. To fix, each of these functions needs the GCC attribute `__attribute__ ((used))`.

-By default, the linker will place global variables in SRAM. This is normally desired behavior, but the process stacks are too large to fit in SRAM. To fix, each of these variables needs the GCC attributes `__attribute__ ((lower)) __attribute__ ((persistent))`.

-Every time a process is pre-empted, its context must be saved to that process's stack within the WDT interrupt. By default, each function has a prologue and epilogue that saves and restores any registers modified by the function. In this case, control flow does not immediately return to the process, so *all* registers need to be saved. We need very careful control over the WDT interrupt, so we use the GCC attribute `__attribute__ ((naked))` to prevent the prologue and epilogue from being generated by the compiler. This was also the first instance where manually writing assembly was required to carefully push all registers to the stack, save the process's stack pointer, and return control flow to ARCOS. This much control was not available with the TI compiler.

LED driver issues:

The handmade LED panel used to demo ARCOS is constructed using WS2812B LEDS, which require a very time-sensitive protocol to communicate. The communication protocol was originally implemented by having the CPU wait for the eUSCI transmit buffer to be empty, then copy the next byte to be transmitted into the transmit buffer. Once the buffer of 9 bytes for the current LED was empty, it would refill the buffer with 9 new bytes for the next LED to be transmitted and repeat until all LEDs were sent. This worked but was only tested on a single color.

This was re-written to use CPU interrupts, where every time the transmit buffer of the eUSCI module was empty, the interrupt would copy the next byte into the transmit buffer. It would then check if the buffer for the current LED was empty, and refill it if needed. This implementation didn't work because the interrupt had to refill the 9-byte buffer for the next LED before the eUSCI module finished sending a single byte, which was not enough time.

This was re-written again so that the led_draw() function would work on filling one 9-byte buffer while an ISR simply copied each byte in a second 9-byte buffer. Once the led_draw() function was finished filling the 9-byte buffer, it would wait for the ISR to finish, then swap the two buffers and start filling the old buffer. This implementation worked for a single channel, but once two channels (one for each half of the panel) were being used, the led_draw() function could not fill the 2 9-byte buffers in the time it took for the ISRs to send them. This was even with the highest level of optimizations enabled in the compiler.

At this point, a great deal of effort went into optimizing the function that actually fills the buffers, led_RGB_to_TX(). Its implementation went through many iterations, but eventually settled on three separate look-up-tables for each byte. It was also inlined to further reduce overhead. This led to a dramatic performance improvement, but the led_draw() function still could not keep up with the ISRs.

Despite all the improvements, performance was still too poor to fill the transmit buffers fast enough. This resulted in another optimization pass, this time looking at the generated assembly, pre-caching as many address calculations as possible, reducing memory overhead, combining duplicate logic for the two channels, and more. Again, this led to a performance improvement, but it was still just a little too slow to function properly.

The issue was resolved with one more major implementation change. This time, all of the previous improvements were combined with the use of the Direct Memory Access (DMA) module. Instead of an ISR copying each byte in both of the transmit buffers to UCBxTXBUF, the DMA controller did it directly. Each of these transfers uses just a few (~4-5) MCLK cycles, meaning the led_draw() function could dedicate nearly all of it's time to filling the transmit buffers while the DMA controller worked in parallel. *Finally*, the led_draw() function could keep up with the eUSCI module. The led_draw() function can now write to both halves of the LED panel simultaneously, at full speed, including converting an RGB framebuffer to GRB and flipping the alternating rows of the panel. The only caveat is that led_draw() disables interrupts while it is running in order to ensure that it can keep up.

Once testing led_draw() finished, it was combined with ARCOS to test that they worked concurrently. Strangely, led_draw() broke once ARCOS tried to run it. Even if led_draw() was the only process, it still would not work. The problem was that, by design, ARCOS uses FRAM for process stacks. Running such performance critical code in FRAM added so much overhead that it could not keep up. A quick modification of ARCOS allowed specifying where the stack is for a particular process, and the led_draw() process was set to use SRAM for its stack. This resolved the problem.

In hindsight, this is the first time I have been forced to write software where high performance is critical to its proper function. In past projects, poor performance just meant it took longer, not that it simply *would not work*. It was quite a fun challenge.

Section 6: Conclusion

A basic RTOS can be implemented on a microcontroller with limited resources, but there are several limitations that impossible to work around without a significant performance impact. Processes must be run out of slower memory, scheduling overhead is not negligible, and additional care must be taken to prevent race conditions. Despite these limitations, even a simple RTOS dramatically reduces software complexity, can increase reliability, and increases code reusability.

Section 7: References

MSP430-specific Resources
MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide:
https://www.ti.com/lit/ug/slau367p/slau367p.pdf
MSP430FR698x(1), MSP430FR598x(1) Datasheet:
https://www.ti.com/lit/ds/symlink/msp430fr6989.pdf
MSP-EXP430FR6989 Launchpad User's Guide:
https://www.ti.com/lit/ug/slau627a/slau627a.pdf
MSP430 Embedded Application Binary Interface (ABI):
https://www.ti.com/lit/an/slaa534a/slaa534a.pdf
MSP430 FRAM How To and Best Practices:
https://www.ti.com/lit/an/slaa628a/slaa628a.pdf
MSP430 FRAM Quality and Reliability:
https://www.ti.com/lit/an/slaa526a/slaa526a.pdf

Misc. Resources
WS2812B Datasheet:
https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf
GCC Extended ASM:
https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
GCC Function Attributes:
https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html
GCC Variable Attributes:
https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

Section 8: Source Code

Source code ZIP download:
https://andrewcourtemanche.com/files/ECE649_FinalProject_Source_Andrew_Courtemanche.zip

main.c

```
/*
Andrew R. Courtemanche 2020/12
*/

#define ARC_MSP_USE_GPIO
#define ARC_MSP_TYPE_msp430fr6989
#include "arc_msp_helper.h"

#include "led_panel.h"

#include "arcos.h"

#include <msp430.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>

//pin defines
#define GREEN_LED &P(9,7)
#define RED_LED &P(1,0)
#define LEFT_BTN &P(1,1)
#define RIGHT_BTN &P(1,2)

__attribute__ ((used))
__attribute__ ((noinline))
void process1(void) {
    while (true) {
        if (digitalRead(RIGHT_BTN) == LOW) {
            digitalWrite(GREEN_LED, HIGH);
        } else {
            digitalWrite(GREEN_LED, LOW);
        }
        arcos_proc_yield();
    }
}

__attribute__ ((used))
__attribute__ ((noinline))
void process2(void) {
    while (true) {
        if (digitalRead(LEFT_BTN) == LOW) {
            digitalWrite(RED_LED, HIGH);
        } else {
            digitalWrite(RED_LED, LOW);
        }
        arcos_proc_yield();
    }
}

//this is too large to fit in SRAM, so it is put in FRAM
__attribute__ ((lower))
__attribute__ ((persistent)) //by default, __attribute__ ((lower)) will place in
SRAM, linking fails if this is not present
uint8_t fb[LED_PANEL_WIDTH*LED_PANEL_HEIGHT*3] = {0};

void fb_clear(void) {
    for(uint16_t i=0; i<sizeof(fb); i++) {
        fb[i] = 0;
    }
}

__attribute__((used))
__attribute__ ((noinline))
void process_render(void) {
    while (true) {
        fb_clear();
        //fill fb with gradient
        for (uint16_t x=0; x<LED_PANEL_WIDTH; x++){
            for (uint16_t y=0; y<LED_PANEL_HEIGHT; y++){
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 0] += x + y;
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 1] += (LED_PANEL_WIDTH - 1 - x)
+ (LED_PANEL_WIDTH - 1 - y);
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 2] += 0;
            }
        }
        arcos_proc_yield();
        led_draw(&fb[0]);
        led_flush();
        for(volatile uint32_t delay = 100000; delay>0; delay--); //adds arbitrary
delay

        fb_clear();
        //fill fb with opposite gradient
        for (uint16_t x=0; x<LED_PANEL_WIDTH; x++){
            for (uint16_t y=0; y<LED_PANEL_HEIGHT; y++){
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 0] += (LED_PANEL_WIDTH - 1 - x)
+ (LED_PANEL_WIDTH - 1 - y);
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 1] += x + y;
                fb[((x + (y*LED_PANEL_WIDTH))*3) + 2] += 0;
            }
        }
        arcos_proc_yield();
        led_draw(&fb[0]);
        led_flush();
        for(volatile uint32_t delay = 100000; delay>0; delay--); //adds arbitrary
delay
    }
}

//place in upper FRAM, not SRAM to keep SRAM clear
__attribute__ ((upper))
struct arcos_proc_s process1_s;
```

```c
__attribute__ ((upper))
struct arcos_proc_s process2_s;
__attribute__ ((upper))
struct arcos_proc_s process_render_s;
__attribute__ ((upper))
struct arcos_proc_s process_startup_s;

__attribute__((used))
__attribute__ ((noinline))
void process_startup(void) {
    //red LED init
    pinMode(RED_LED, MODE_OUTPUT);
    digitalWrite(RED_LED, LOW);

    //green LED init
    pinMode(GREEN_LED, MODE_OUTPUT);
    digitalWrite(GREEN_LED, LOW);

    //left BTN init
    pinMode(LEFT_BTN, MODE_INPUT_PULLUP);

    //right BTN init
    pinMode(RIGHT_BTN, MODE_INPUT_PULLUP);

    //zero out RAM, nothing should be using it. This is mostly just as a
demonstration that this is running within FRAM now.
    //0x001C00-0x0023FF RAM
    for (uint16_t i=0x1C00; i<=0x0023FF; i++) {
        *((uint8_t *)i) = 0;
    }

    arcos_proc_create(&process1_s, &process1, 0, 100); //automatic stack
allocation, 100 priority
    arcos_proc_start(&process1_s);
    arcos_proc_create(&process2_s, &process2, 0, 100); //automatic stack
allocation, 100 priority
    arcos_proc_start(&process2_s);
    arcos_proc_create(&process_render_s, &process_render, 0x2400, 100); //place
this process in SRAM (0x2400 is the top of SRAM), 100 priority
    arcos_proc_start(&process_render_s);
    //Here, this process returns and terminates. It will not run again.
}

void main(void) {
    arcos_init();
    arc_msp_setup();
    led_init();

    arcos_proc_create(&process_startup_s, &process_startup, 0, 0); //automatic
stack allocation, 0 (maximum) priority
    arcos_proc_start(&process_startup_s);

    arcos_start();
}
```

arcos_config.h

```c
/*
Andrew R. Courtemanche 2020/12
*/

#ifndef ARCOS_CONFIG_GUARD
#define ARCOS_CONFIG_GUARD

#ifndef __MSP430FR6989__
    #error No supported microcontroller detected.
#endif

#ifndef ARCOS_CONFIG_TIMESLICE_MICROSECONDS
    #define ARCOS_CONFIG_TIMESLICE_MICROSECONDS (1000) //1ms //UNUSED
#endif
#ifndef ARCOS_CONFIG_PROC_COUNT_MAX
    #define ARCOS_CONFIG_PROC_COUNT_MAX (8)
#endif
#ifndef ARCOS_CONFIG_PROC_STACK_SIZE_MAX
    #define ARCOS_CONFIG_PROC_STACK_SIZE_MAX (1024)
#endif

#ifdef __MSP430FR6989__
    #define ARCOS_CONFIG_RAM_SIZE (2048)

    #define ARCOS_CONFIG_KERNEL_SIZE_STACK (512)

    #define ARCOS_CONFIG_REG_COUNT (16)
    #define ARCOS_CONFIG_REG_SIZE (2)

    #define ARCOS_CONFIG_CLOCK_SRC_FREQ_LFXT (32768)
    #define ARCOS_CONFIG_CLOCK_SRC_FREQ_HFXT (0)
    #define ARCOS_CONFIG_CLOCK_SRC_FREQ_VLO (10000)
    #define ARCOS_CONFIG_CLOCK_SRC_FREQ_MOD (5000000)
    #define ARCOS_CONFIG_CLOCK_SRC_FREQ_LFMOD (ARCOS_CONFIG_CLOCK_FREQ_MOD/128)
#endif

#endif //end ARCOS_CONFIG_GUARD
```

arcos.h

```c
/*
Andrew R. Courtemanche 2020/11
Updated 2020/12
*/

/*
RTC_C
32,768Hz LFXT crystal
2,048 bytes RAM
```

```
48,000 bytes 16-bit address FRAM
81,920 bytes 20-bit address FRAM
defaults:
    MCLK DCO 1MHz
    SMCLK DCO 1MHz
    ACLK REFO(?) 32,768 Hz
DCO:
    (24MHz / 21MHz / 16MHz / 8MHz / 7MHz / 5.33MHz / 4MHz / 3.5MHz / 2.67 MHz /
1MHz)
    1MHz
    2.67MHz
    3.5MHz
    4MHz
    5.33MHz
    7MHz
    8MHz
    16MHz
    21MHz
    24MHz
source dividers for MCLK, SMCLK, ACLK:
    1
    2
    4
    8
    16
    32
MCLK sources:
    LFXTCLK  (32768Hz / 16384Hz / 8192Hz / 4096Hz / 2048Hz / 1024Hz)
    VLOCLK   (10kHz / 5kHz / 2.5kHz / 1.25kHz / ~625Hz / 312.5Hz)
    LFMODCLK (~39kHz / ~19.5kHz / ~9.77kHz / ~4.88kHz / ~2.44kHz / ~1.22kHz)
    DCOCLK   (configurable)
    MODCLK   (5MHz / 2.5MHz / 1.25MHz / ~625kHz / ~312.6kHz / ~156.25kHz)
    HFXTCLK  (Not available)
SMCLK sources:
    LFXTCLK  (32768Hz / 16384Hz / 8192Hz / 4096Hz / 2048Hz / 1024Hz)
    VLOCLK   (10kHz / 5kHz / 2.5kHz / 1.25kHz / ~625Hz / 312.5Hz)
    LFMODCLK (~39kHz / ~19.5kHz / ~9.77kHz / ~4.88kHz / ~2.44kHz / ~1.22kHz)
    DCOCLK   (configurable)
    MODCLK   (5MHz / 2.5MHz / 1.25MHz / ~625kHz / ~312.6kHz / ~156.25kHz)
    HFXTCLK  (Not available)
ACLK sources:
    LFXTCLK  (32768Hz / 16384Hz / 8192Hz / 4096Hz / 2048Hz / 1024Hz)
    VLOCLK   (10kHz / 5kHz / 2.5kHz / 1.25kHz / ~625Hz / ~312.5Hz)
    LFMODCLK (~39kHz / ~19.5kHz / ~9.77kHz / ~4.88kHz / ~2.44kHz / ~1.22kHz)

WDT sources:
    SMCLK
    ACLK
    VLOCLK (10kHz)
WDT intervals:
    CLK/2^31 (18:12:16 at 32.768kHz)
    CLK/2^27 (01:08:16 at 32.768kHz)
    CLK/2^23 (00:04:16 at 32.768kHz)
    CLK/2^19 (00:00:16 at 32.768kHz)
    CLK/2^15 (1 s at 32.768kHz)
    CLK/2^13 (250ms at 32.768kHz)
    CLK/2^9  (15.625ms at 32.768kHz
    CLK/2^6  (1.95ms at 32.768kHz)

*/

/*
2,048 bytes RAM (fastest):
0x001C00-0x0023FF - general space
    0x001C00-0x00.... - global variables
    0x00....-0x00.... - 1024 bytes process contexts (32 contexts of 16 2-byte
registers)
    0x00....-0x0023FA - OS stack
    0x0023FC-0x0023FD - main() return pointer
    0x0023FE-0x0023FF - NULL

48,000 bytes 16-bit address FRAM:
0x004400-0x00FF7F - general space
    0x004400-0x007F7F 15,232 bytes - code / data
    0x007F80-0x00FF7F 32,768 bytes - arcos process stacks (32x1024)
0x00FF80-0x00FFFF - interrupt vectors and signatures

81,920 bytes 20-bit address FRAM (slowest):
0x010000-0x023FFFF - general space
    0x010000-0x0..... - code
    0x0.....-0x023FFF - heap

 */

#ifndef ARCOS_GUARD
#define ARCOS_GUARD

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

#include "arcos_config.h"

//possible states of a process
enum arcos_proc_status_e {
    PROC_STATE_UNINITIALIZED = 0,
    PROC_STATE_TERMINATED,
    PROC_STATE_STOPPED,
    //PROC_STATE_SUSPENDED, //future use
    //PROC_STATE_BLOCKED, //future use
    PROC_STATE_READY,
    PROC_STATE_RUNNING,
};

//describes a particular process
//included in header so size is known
struct arcos_proc_s {
    uint8_t priority;
    enum arcos_proc_status_e status;
    uint16_t SP;
    void (*callback)(void);
```

```c
};

//marks process as ready for execution
void arcos_proc_start(struct arcos_proc_s * handle);

//terminates specified process
void arcos_proc_terminate(struct arcos_proc_s * handle);

//yields timeslice to another process
void arcos_proc_yield(void);

//initializes a arcos_proc_s struct and internal ARCOS variables
//0 is highest priority, 255 is lowest priority
void arcos_proc_create(struct arcos_proc_s * handle, void (*callback)(void),
uint16_t SP, uint8_t priority);

//initial configuration of the core, sets up watchdog, clocks, timers, etc.
//should be called immediately after boot
void arcos_init(void);

//hands execution over to ARCOS
//will never return, and current stack is invalidated
void arcos_start(void);

#endif //end ARCOS_GUARD
```

arcos.c

```c
/*
Andrew R. Courtemanche 2020/11
Updated 2020/12
*/

#include "arcos.h"

#include <msp430.h>
#include <string.h>

//FOR FUTURE USE
//provides a valid ISR stub for all interrupt sources
__attribute__ ((interrupt(AES256_VECTOR)))
__attribute__ ((interrupt(RTC_VECTOR)))
__attribute__ ((interrupt(LCD_C_VECTOR)))
__attribute__ ((interrupt(PORT4_VECTOR)))
__attribute__ ((interrupt(PORT3_VECTOR)))
__attribute__ ((interrupt(TIMER3_A1_VECTOR)))
__attribute__ ((interrupt(TIMER3_A0_VECTOR)))
__attribute__ ((interrupt(PORT2_VECTOR)))
__attribute__ ((interrupt(TIMER2_A1_VECTOR)))
__attribute__ ((interrupt(TIMER2_A0_VECTOR)))
__attribute__ ((interrupt(PORT1_VECTOR)))
__attribute__ ((interrupt(TIMER1_A1_VECTOR)))
__attribute__ ((interrupt(TIMER1_A0_VECTOR)))
__attribute__ ((interrupt(DMA_VECTOR)))
__attribute__ ((interrupt(USCI_B1_VECTOR)))
__attribute__ ((interrupt(USCI_A1_VECTOR)))
__attribute__ ((interrupt(TIMER0_A1_VECTOR)))
__attribute__ ((interrupt(TIMER0_A0_VECTOR)))
__attribute__ ((interrupt(ADC12_VECTOR)))
__attribute__ ((interrupt(USCI_B0_VECTOR)))
__attribute__ ((interrupt(USCI_A0_VECTOR)))
__attribute__ ((interrupt(ESCAN_IF_VECTOR)))
//__attribute__ ((interrupt(WDT_VECTOR)))
__attribute__ ((interrupt(TIMER0_B1_VECTOR)))
__attribute__ ((interrupt(TIMER0_B0_VECTOR)))
__attribute__ ((interrupt(COMP_E_VECTOR)))
__attribute__ ((interrupt(UNMI_VECTOR)))
__attribute__ ((interrupt(SYSNMI_VECTOR)))
__attribute__ ((interrupt))
static void arcos_isr_stub(void) {
    return;
}

struct arcos_kernel_s {
    uint16_t SP;
    uint8_t stack[ARCOS_CONFIG_KERNEL_SIZE_STACK];
    struct arcos_proc_s * proc_current;
    uint16_t proc_count;
    struct arcos_proc_s * proc_list[ARCOS_CONFIG_PROC_COUNT_MAX];
    uint8_t
proc_stack[ARCOS_CONFIG_PROC_COUNT_MAX][ARCOS_CONFIG_PROC_STACK_SIZE_MAX];
};

//stores all information that the kernel needs
__attribute__ ((lower))
__attribute__ ((persistent))
static struct arcos_kernel_s arcos_var_kernel = {0};

//These functions are simply intended to increase code readability
//  and reduce potential mistakes. They have the inline specifier to hint that they
//  should probably not result in a true function call.
static inline void WDT_restop(void) {
    WDTCTL = WDTPW | WDTHOLD | WDTCNTCL | WDTCTL_L;
}
static inline void WDT_stop(void) {
    WDTCTL = WDTPW | WDTHOLD | WDTCTL_L;
}
static inline void WDT_reset(void) {
    WDTCTL = WDTPW | WDTCNTCL | WDTCTL_L;
}
static inline void WDT_restart(void) {
    WDTCTL = WDTPW | WDTCNTCL | (WDTCTL_L & (~WDTHOLD));
}
static inline void WDT_start(void) {
    WDTCTL = WDTPW | (WDTCTL_L & (~WDTHOLD));
}

//Triggers a reset and PUC, hard restarting the entire MCU
```

```
__attribute__ ((used))
__attribute__ ((noreturn))
__attribute__ ((naked))
static void arcos_os_pwr_reset(void) {
    WDTCTL = WDTPW | WDTSSEL__ACLK | WDTIS__64; //sets WDT to watchdog mode
    WDTCTL = 0xFF | WDTSSEL__ACLK | WDTIS__64; //intentionally writes incorrect
password to WDTCTL to trigger a hard reset
    while(true); //wait
}

//simple priority round-robin scheduling
//WARNING: high priority tasks can starve lower priority tasks
__attribute__ ((noreturn))
__attribute__ ((naked))
static void arcos_os_schedule(void) {
    if (arcos_var_kernel.proc_count > 0) { //are there any processes left?
        for (uint8_t i=0; i<arcos_var_kernel.proc_count; i++) { //read through the
list of processes
            if (arcos_var_kernel.proc_list[i]->status == PROC_STATE_READY) { //get
the first ready process, this will be the highest priority one since list is sorted
                arcos_var_kernel.proc_current = arcos_var_kernel.proc_list[i];
//set as current process
                uint8_t ii = i+1;
                //move this process to the back of the list of processes with the
same priority
                for (; (arcos_var_kernel.proc_list[ii]->priority ==
arcos_var_kernel.proc_current->priority) && (ii < arcos_var_kernel.proc_count);
ii++) {
                    arcos_var_kernel.proc_list[ii-1] =
arcos_var_kernel.proc_list[ii];
                }
                arcos_var_kernel.proc_list[ii-1] = arcos_var_kernel.proc_current;
                break;
            }
        }
    } else {
        arcos_os_pwr_reset(); //There are no more processes left to schedule. This
is assumed to be a mistake, so restart the MCU
    }

    __asm(" BIC.B %0, %1\n"::"r"(WDTIFG), "r"(SFRIFG1_L)); //clear WDTIFG using
BIC.B, as recommended in User's Guide page 74
    WDT_restart(); //restart the WDT
    //_enable_interrupts(); //not needed because RETI will restore SR and enable
interrupts
    arcos_var_kernel.proc_current->status = PROC_STATE_RUNNING; //mark the selected
process as running
    __set_SP_register(arcos_var_kernel.proc_current->SP); //change the stack
pointer to the process
    __asm(" POPM.A #12,R15\n"); //restore context
    __asm(" RETI\n"); //return control flow to process
}

//kernel entry point after startup, process pre-emption, etc
__attribute__ ((noreturn))
__attribute__ ((naked))
static void arcos_os_run(void) {
    WDT_stop(); //stop the WDT
    //other features added here
    arcos_os_schedule(); //schedule the next process
}

//ISR called after timeslice expires
//saves context of current process and returns control flow to kernel
__attribute__ ((interrupt(WDT_VECTOR)))
__attribute__ ((naked))
static void arcos_os_isr_timeout_slice(void) {
    //PC and SR are already pushed by the interrupt
    //Pushes R4-R15 to the stack, saving all 20 bits. Each register uses 4 bytes,
so total size is 48bytes
    __asm(" PUSHM.A #12,R15\n");
    //At this point, process context is saved except SP
    uint16_t proc_SP = __get_SP_register(); //get SP
    arcos_var_kernel.proc_current->SP = proc_SP; //save process SP
    arcos_var_kernel.proc_current->status = PROC_STATE_READY; //mark process as
ready to run

    __set_SP_register(arcos_var_kernel.SP); //change stack pointer to kernel
    __asm(" PUSHX.A %0\n"::"r"(&arcos_os_run)); //push address of arcos_os_run() to
stack
    __asm(" RETA\n"); //pop address and return control flow to kernel
}

//sorts processes by priority
//very crappy sorting algorithm
static inline void arcos_os_proc_sort(void) {
    for (uint8_t i=0; i<arcos_var_kernel.proc_count; i++) {
        struct arcos_proc_s * temp = arcos_var_kernel.proc_list[i];
        int16_t ii = i - 1;
        for (; ii >= 0 && arcos_var_kernel.proc_list[ii]->priority > temp-
>priority; ii--) {
            arcos_var_kernel.proc_list[ii+1] = arcos_var_kernel.proc_list[ii];
        }
        arcos_var_kernel.proc_list[ii+1] = temp;
    }
}

//removes process from process list
void arcos_proc_terminate(struct arcos_proc_s * handle) {
    uint16_t GIE_BACKUP = _get_SR_register() & GIE; //store GIE
    __asm(" DINT \n NOP \n"); //disable interrupts

    handle->status = PROC_STATE_TERMINATED;

    //finds process index from pointer
    for (uint8_t i=0; i<arcos_var_kernel.proc_count; i++) {
        if (arcos_var_kernel.proc_list[i] == handle) {
            arcos_var_kernel.proc_list[i] =
arcos_var_kernel.proc_list[arcos_var_kernel.proc_count-1];
        }
    }
    arcos_var_kernel.proc_count--;
    arcos_os_proc_sort();

    __asm(" BIS.B %0, SR \n NOP \n"::"r"(GIE_BACKUP)); //restore GIE
}

//runs when a process returns
__attribute__ ((noreturn))
__attribute__ ((naked))
static void arcos_os_proc_return(void) {
    _disable_interrupts();

    arcos_proc_terminate(arcos_var_kernel.proc_current);

    __set_SP_register(arcos_var_kernel.SP);
    __asm(" PUSHX.A %0\n"::"r"(&arcos_os_run));
    __asm(" RETA\n");
}

//initial kernel entry point
__attribute__ ((noreturn))
__attribute__ ((naked))
void arcos_start(void) {
    _disable_interrupts();

    arcos_var_kernel.SP = (uint16_t) arcos_var_kernel.stack +
sizeof(arcos_var_kernel.stack); //valid cast because we know the stacks are in the
lower 64K of memory

    __set_SP_register(arcos_var_kernel.SP);
    __asm(" PUSHX.A %0\n"::"r"(&arcos_os_run));
    __asm(" RETA\n");
}

//initial configuration of the core, sets up watchdog, clocks, timers, etc.
//should be called immediately after boot
void arcos_init(void) {
    _disable_interrupts();
    WDTCTL = WDTPW | WDTHOLD | WDTSSEL__ACLK | WDTTMSEL | WDTCNTCL | WDTIS__64; //
Configure WDT
    __asm(" BIC.B %0, %1\n"::"r"(WDTIFG), "r"(SFRIFG1_L)); //clear WDTIFG using
BIC.B, as recommended in User's Guide page 74
    SFRIE1 = WDTIE; //enable WDT interrupt

    FRCTL0 = FRCTLPW; //unlock FRCTL registers
    FRCTL0_L = NWAITS_1; //set FRAM wait mode to 1 cycle (max FRAM access frequency
is 8Mhz, we are setting CPU to 16Mhz)
    FRCTL0_H = 0; //lock FRCTL registers

    CSCTL0 = CSKEY; //unlock CSCTLx registers
    CSCTL1 = DCORSEL | DCOFSEL_4; //set DCO to 16Mhz
    CSCTL2 = SELA__LFXTCLK | SELS__MODCLK | SELM__DCOCLK; //select clock sources
    CSCTL3 = DIVA__1 | DIVS__2 | DIVM__1; //AUX 32768Hz, SM 2.5Mhz, M 16Mhz
    //CSCTL2 = SELA__LFXTCLK | SELS__DCOCLK | SELM__DCOCLK; //select clock sources
    //CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1; //AUX 32768Hz, SM 16Mhz, M 16Mhz
    //CSCTL3 = (CSCTL3 & ~((DIVS0 | DIVS1 | DIVS2) | (DIVM0 | DIVM1 | DIVM2))) |
(DIVS_0 | DIVM_0); //set SMCLK and MCLK dividers to /1
    CSCTL0_H = 0; //lock CSCTLx registers

    //initialize unused ports to make compiler shut up
    PADIR = 0x00;
    PAOUT = 0x00;
    PBDIR = 0x00;
    PBOUT = 0x00;
    PCDIR = 0x00;
    PCOUT = 0x00;
    PDDIR = 0x00;
    PDOUT = 0x00;
    PEDIR = 0x00;
    PEOUT = 0x00;
}

//yields timeslice to another process by immediately setting WDT interrupt flag
inline void arcos_proc_yield(void) {
    //SFRIFG1 = SFRIFG1 | WDTIFG;
    __asm(" BIS.B %0, %1 \n NOP \n"::"i"(WDTIFG), "r"(SFRIFG1));
}

//marks process as ready to run
void arcos_proc_start(struct arcos_proc_s * handle) {
    uint16_t GIE_BACKUP = _get_SR_register() & GIE; //store GIE
    __asm(" DINT \n NOP \n"); //disable interrupts

    handle->status = PROC_STATE_READY; //TODO: add error checking

    __asm(" BIS.B %0, SR \n NOP \n"::"r"(GIE_BACKUP)); //restore GIE
}

/*
 * KNOWN ISSUE: Stack selection is based on number of processes, if a process ever
terminates and a new one is created, stack pointers will then overlap
 */
//0 is highest priority, 255 is lowest priority
//initializes a arcos_proc_s struct and internal ARCOS variables
void arcos_proc_create(struct arcos_proc_s * handle, void (*callback)(void),
uint16_t SP, uint8_t priority) {
    uint16_t GIE_BACKUP = _get_SR_register() & GIE; //store GIE
    __asm(" DINT \n NOP \n"); //disable interrupts

    //initialize arcos_proc_s struct
    handle->priority = priority;
    handle->status = PROC_STATE_STOPPED;
    if (SP) {
        handle->SP = SP; //use specific stack pointer, if provided
    } else {
        handle->SP = (uint16_t)
arcos_var_kernel.proc_stack[arcos_var_kernel.proc_count] +
ARCOS_CONFIG_PROC_STACK_SIZE_MAX; //assign a process stack
    }
    handle->callback = callback;

    //manually manipulate process stack
    handle->SP -= 4;
    *((uintptr_t *)handle->SP) = (uintptr_t) &arcos_os_proc_return; //push return
address of arcos_os_proc_return()
    handle->SP -= 2;
    *((uint16_t *)handle->SP) = ((uint32_t)callback) & 0x0000FFFF; //push process
PC
    handle->SP -= 2;
    *((uint16_t *)handle->SP) = ((((uint32_t)callback) & 0x000F0000) >> 4) |
(0b00001000 & 0x1FF); //push process SR
    handle->SP -= 4 * 12; //push 12 empty registers

    arcos_var_kernel.proc_list[arcos_var_kernel.proc_count] = handle; //add pointer
to process list
    arcos_var_kernel.proc_count++;
    arcos_os_proc_sort();

    __asm(" BIS.B %0, SR \n NOP \n"::"r"(GIE_BACKUP)); //restore GIE
}
```

led_panel.h

```
/*
Andrew R. Courtemanche 2020/12
*/

#ifndef LED_PANEL_GUARD
#define LED_PANEL_GUARD

#include <stdint.h>

#define LED_PANEL_WIDTH 32
#define LED_PANEL_HEIGHT 32
#define LED_CHANNEL_WIDTH 32
#define LED_CHANNEL_HEIGHT 16

//initialize required registers
void led_init(void);

//make sure LED strips have enough time to reset
void led_flush(void);

//draw given 32x32 24 bit RGB framebuffer
void led_draw(uint8_t * rgb_buf);

#endif //end include guard
```

led_panel.c

```
/*
```

```
Andrew R. Courtemanche 2020/12
*/

/*
This file uses the eUSCI module configured as an SPI master to output the very
timing sensitive protocol
the WS2812B individually addressable LEDs use.

The SPI module is configured with a 2.5MHz bit clock, which means a bit is shifted
out every 0.4us.
Each bit is encoded as a 1.2us signal split into one high period and one low
period. The ratio
between these high and low periods encode a single bit.

To encode a 0 bit: 0.4us high, 0.8us low
To encode a 1 bit: 0.8us high, 0.4us low

Since a bit is shifted out every 0.4us, we can encode these signals as a series of
bits.
Each color bit is encoded as three bits for transmission. A 0 bit is encoded as
0b100
and a 1 bit is encoded as 0b110. This means a single color byte is encoded as three
bytes
for transmission. Furthermore, a full RGB color is encoded as a series of 9 bytes.

For example:
11001010 would be encoded as:
11011010 01001101 00110100

For example:
RGB 11111111 00000000 10101010 would be encoded as:
10010010 01001001 00100100 11011011 01101101 10110110 11010011 01001101 00110100
G        G        G        R        R        R        B        B        B
Keep in mind RGB is converted to GRB for transmission.
*/

#include "led_panel.h"

#define ARC_MSP_USE_GPIO
#define ARC_MSP_TYPE_msp430fr6989
#include "arc_msp_helper.h"

#include <msp430.h>
#include <stdint.h>
#include <stdbool.h>

//LUT for first TX byte
__attribute__ ((lower))
const uint8_t led_TX_LUT0[] = {
  0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92,
  0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92,
  0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x93, 0x93, 0x93, 0x93,
  0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93,
  0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93, 0x93,
  0x93, 0x93, 0x93, 0x93, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a,
  0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a,
  0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a, 0x9a,
  0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b,
  0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b,
  0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0x9b, 0xd2, 0xd2, 0xd2, 0xd2,
  0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2,
  0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2, 0xd2,
  0xd2, 0xd2, 0xd2, 0xd2, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3,
  0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3,
  0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3, 0xd3,
  0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda,
  0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda,
  0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xda, 0xdb, 0xdb, 0xdb, 0xdb,
  0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb,
  0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb,
  0xdb, 0xdb, 0xdb, 0xdb
};

//LUT for second TX byte
__attribute__ ((lower))
const uint8_t led_TX_LUT1[] = {
  0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d,
  0x4d, 0x4d, 0x4d, 0x4d, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69,
  0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x49, 0x49, 0x49, 0x49,
  0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d,
  0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x6d, 0x6d, 0x6d, 0x6d,
  0x6d, 0x6d, 0x6d, 0x6d, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49,
  0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x69, 0x69, 0x69, 0x69,
  0x69, 0x69, 0x69, 0x69, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d,
  0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d,
  0x4d, 0x4d, 0x4d, 0x4d, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69,
  0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x49, 0x49, 0x49, 0x49,
  0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d,
  0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x6d, 0x6d, 0x6d, 0x6d,
  0x6d, 0x6d, 0x6d, 0x6d, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49,
  0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x69, 0x69, 0x69, 0x69,
  0x69, 0x69, 0x69, 0x69, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d,
  0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d,
  0x4d, 0x4d, 0x4d, 0x4d, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69,
  0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x6d, 0x49, 0x49, 0x49, 0x49,
  0x49, 0x49, 0x49, 0x49, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d, 0x4d,
  0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x69, 0x6d, 0x6d, 0x6d, 0x6d,
  0x6d, 0x6d, 0x6d, 0x6d
};

//LUT for third TX byte
__attribute__ ((lower))
const uint8_t led_TX_LUT2[] = {
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6,
  0x24, 0x26, 0x34, 0x36, 0xa4, 0xa6, 0xb4, 0xb6, 0x24, 0x26, 0x34, 0x36,
  0xa4, 0xa6, 0xb4, 0xb6
};

//unused function, was used for early implementations with a different set of LUTs
//inline void led_byte_to_TX(uint8_t * buf, uint8_t col) {
//    buf[0] = led_TX_LUT0[((col & 0b11100000) >> 5)];
//    buf[1] = led_TX_LUT1[((col & 0b00011000) >> 3)];
//    buf[2] = (led_TX_LUT0[col & 0b00000111] << 1);
//
//    buf[0] = led_TX_LUT0[(col & 0b11100000) >> 5];
//    buf[1] = led_TX_LUT1[(col & 0b00011000) >> 3];
//    buf[2] = led_TX_LUT2[(col & 0b00000111)];
//
//    buf[0] = led_TX_LUT[col*3];
//    buf[1] = led_TX_LUT[col*3+1];
//    buf[2] = led_TX_LUT[col*3+2];
//}

//converts a 24-bit RGB value into a 9-byte GRB encoded stream for transmission
//uses LUTs for performance
static inline void led_RGB_to_TX(uint8_t * buf, uint8_t * rgb) {
//    led_byte_to_TX(buf, rgb[1]);
//    led_byte_to_TX(buf+3, rgb[0]);
//    led_byte_to_TX(buf+6, rgb[2]);

//    buf[0] = led_TX_LUT0[(rgb[1] & 0b11100000) >> 5];
//    buf[3] = led_TX_LUT0[(rgb[0] & 0b11100000) >> 5];
//    buf[6] = led_TX_LUT0[(rgb[2] & 0b11100000) >> 5];
//    buf[1] = led_TX_LUT1[(rgb[1] & 0b00011000) >> 3];
//    buf[4] = led_TX_LUT1[(rgb[0] & 0b00011000) >> 3];
//    buf[7] = led_TX_LUT1[(rgb[2] & 0b00011000) >> 3];
//    buf[2] = led_TX_LUT2[(rgb[1] & 0b00000111)];
//    buf[5] = led_TX_LUT2[(rgb[0] & 0b00000111)];
//    buf[8] = led_TX_LUT2[(rgb[2] & 0b00000111)];

    buf[0] = led_TX_LUT0[rgb[1]];
    buf[3] = led_TX_LUT0[rgb[0]];
    buf[6] = led_TX_LUT0[rgb[2]];

    buf[1] = led_TX_LUT1[rgb[1]];
    buf[4] = led_TX_LUT1[rgb[0]];
    buf[7] = led_TX_LUT1[rgb[2]];

    buf[2] = led_TX_LUT2[rgb[1]];
    buf[5] = led_TX_LUT2[rgb[0]];
    buf[8] = led_TX_LUT2[rgb[2]];
}

//make sure LED strips have enough time to reset
//writes a bunch of zero bytes to panel
void led_flush(void) {
    for (uint16_t i=0; i<100; i++) {
        while (!(UCB0IFG & UCTXIFG));
        UCB0TXBUF = 0;
        while (!(UCB1IFG & UCTXIFG));
        UCB1TXBUF = 0;
    }
}

//draw given 32x32 24 bit RGB framebuffer
void led_draw(uint8_t * rgb_buf) {
    uint16_t GIE_BACKUP = _get_SR_register() & GIE; //store GIE
    __asm(" DINT \n NOP \n"); //disable interrupts

    while (!(UCB0IFG & UCTXIFG)); //make sure nothing is being transmitted already
    while (!(UCB1IFG & UCTXIFG)); //make sure nothing is being transmitted already

    uint_fast8_t tx_buf_select = 0;
    uint8_t tx_buf[2][2][9]; //[channel][buffer][data]

    //do address calculation ahead of time for performance
    uintptr_t tx_ch0_buf0_ptr = (uintptr_t) &tx_buf[0][0][0];
    uintptr_t tx_ch0_buf1_ptr = (uintptr_t) &tx_buf[0][1][0];
    uintptr_t tx_ch1_buf0_ptr = (uintptr_t) &tx_buf[1][0][0];
    uintptr_t tx_ch1_buf1_ptr = (uintptr_t) &tx_buf[1][1][0];

    //initialize transmission buffers
    led_RGB_to_TX((uint8_t *) tx_ch0_buf0_ptr, &rgb_buf[0]);
    led_RGB_to_TX((uint8_t *) tx_ch1_buf0_ptr, &rgb_buf[0 +
((LED_CHANNEL_WIDTH*LED_CHANNEL_HEIGHT)*3)]);

    //set DMA src address to correct buffers
    DMA0SA = tx_ch0_buf0_ptr;
    DMA1SA = tx_ch1_buf0_ptr;

    //enable DMA0 and provide rising edge to kick it off
    DMA0CTL |= DMAEN;
    UCB0IFG &= ~UCTXIFG;
    UCB0IFG |=  UCTXIFG;

    //enable DMA1 and provide rising edge to kick it off
    DMA1CTL |= DMAEN;
    UCB1IFG &= ~UCTXIFG;
    UCB1IFG |=  UCTXIFG;

    volatile uint16_t x=1; //it breaks if this is not volatile
    for (volatile uint16_t y=0; y<LED_CHANNEL_HEIGHT; y++) { //it breaks if this is
not volatile
        for (; x<LED_CHANNEL_WIDTH; x++) {
            tx_buf_select = (tx_buf_select == 0) ? 1 : 0; //switch active buffer

            uintptr_t tx_ch0_buf_temp_ptr;
            uintptr_t tx_ch1_buf_temp_ptr;

            //store address of transmission buffers
            if (tx_buf_select == 0) {
                tx_ch0_buf_temp_ptr = tx_ch0_buf0_ptr;
                tx_ch1_buf_temp_ptr = tx_ch1_buf0_ptr;
            } else {
                tx_ch0_buf_temp_ptr = tx_ch0_buf1_ptr;
                tx_ch1_buf_temp_ptr = tx_ch1_buf1_ptr;
            }

            //set DMA src address to correct buffers
            //this can be done while the DMA is running because it internally
copies these addresses to temporary registers
            DMA0SA = tx_ch0_buf_temp_ptr;
            DMA1SA = tx_ch1_buf_temp_ptr;

            //fill the transmit buffers with the next 18 bytes to be transmitted,
flipping the odd rows because of the way the LED panel is constructed
            if ((y & 0x1) == 0) { //even row
                led_RGB_to_TX((uint8_t *) tx_ch0_buf_temp_ptr,
&rgb_buf[(x+(y*LED_CHANNEL_WIDTH)) * 3]);
                led_RGB_to_TX((uint8_t *) tx_ch1_buf_temp_ptr,
&rgb_buf[((x+(y*LED_CHANNEL_WIDTH)) * 3) +
((LED_CHANNEL_WIDTH*LED_CHANNEL_HEIGHT)*3)]);
            } else { //odd row
                led_RGB_to_TX((uint8_t *) tx_ch0_buf_temp_ptr,
&rgb_buf[((LED_CHANNEL_WIDTH-1-x)+(y*LED_CHANNEL_WIDTH)) * 3]);
                led_RGB_to_TX((uint8_t *) tx_ch1_buf_temp_ptr,
&rgb_buf[(((LED_CHANNEL_WIDTH-1-x)+(y*LED_CHANNEL_WIDTH)) * 3) +
((LED_CHANNEL_WIDTH*LED_CHANNEL_HEIGHT)*3)]);
            }

            //wait for DMA to finish, restart it immediately
            while (DMA0CTL & DMAEN);
            DMA0CTL |= DMAEN;
            while (DMA1CTL & DMAEN);
            DMA1CTL |= DMAEN;
        }
        x=0;
    }

    //wait for DMA to finish, send a 0 to start resetting the panel
    while (DMA0CTL & DMAEN);
    UCB0TXBUF = 0;
    while (DMA1CTL & DMAEN);
    UCB1TXBUF = 0;

    __asm(" NOP \n");
    //_enable_interrupts();
    __asm(" BIS.B %0, SR \n NOP \n"::"r"(GIE_BACKUP)); //restore GIE
}

//initialize required registers
void led_init(void) {
    //P1.6 UCB0SIMO
    //P4.0 UCB1SIMO
    arc_msp_setup(); //setup GPIO

    //SPI config
    /*
    https://www.ti.com/lit/ug/slau627a/slau627a.pdf
    page 17:
    P1.6 UCB0SIMO
    P4.0 UCB1SIMO
    */
    pinFunc(&P(1,6), 1); //configure pin functions
    pinFunc(&P(4,0), 2); //configure pin functions

    UCB0BRW = 0; //do not divide BRCLK
    UCB1BRW = 0; //do not divide BRCLK

    UCB0CTLW0 = UCMSB | UCMST | UCSYNC | UCSSEL__SMCLK; //MSB first, Master mode,
SPI, use SMCLK as clock source
    UCB1CTLW0 = UCMSB | UCMST | UCSYNC | UCSSEL__SMCLK; //MSB first, Master mode,
SPI, use SMCLK as clock source
    //end SPI config

    //DMA config
    DMACTL0 = DMA0TSEL__UCB0TXIFG0 | DMA1TSEL__UCB1TXIFG0; //select DMA triggers
    //DMACTL4 = ROUNDROBIN;

    DMA0CTL = DMADT_0 | DMASRCINCR_3 | DMADSTBYTE | DMASRCBYTE; //single transfer
mode, increment source address, byte destination, byte source
    DMA1CTL = DMADT_0 | DMASRCINCR_3 | DMADSTBYTE | DMASRCBYTE; //single transfer
mode, increment source address, byte destination, byte source

    DMA0SA = 0; //initialize source address to zero, will be changed later
    DMA1SA = 0; //initialize source address to zero, will be changed later

    DMA0DA = 0x0640 + 0x000E; //UCB0TXBUF as destination address
    DMA1DA = 0x0680 + 0x000E; //UCB1TXBUF as destination address

    DMA0SZ = 9; //transfer 9 bytes
    DMA1SZ = 9; //transfer 9 bytes
}
```

arc_msp_helper.h

```
/*
Andrew R. Courtemanche 2020/11
Updated 2020/12
*/
```

```c
#ifndef ARC_MSP_HELPER_GUARD
#define ARC_MSP_HELPER_GUARD

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

//For future use to support different MSP430 variants
#ifdef ARC_MSP_TYPE_msp430fr6989
    #ifdef ARC_MSP_TYPE
        #error Multiple MSP types defined
    #endif //end ARC_MSP_TYPE
    #define ARC_MSP_TYPE
#endif //end ARC_MSP_TYPE_msp430fr6989
#ifndef ARC_MSP_TYPE
    #error No MSP type defined
#endif //end ARC_MSP_TYPE

#ifdef ARC_MSP_USE_ALL
    #define ARC_MSP_USE_GPIO
#endif //end ARC_MSP_USE_ALL

//Convenience defines
#define NULL 0

#define LOW 0
#define HIGH 1

#define DISABLE 0
#define ENABLE 1

#define RISING_EDGE 0
#define FALLING_EDGE 1
//end convenience defines

#ifdef ARC_MSP_USE_GPIO

    //PxDIR is 0 for input
    //PxDIR is 1 for output
    //PxOUT is 0 for pull-down
    //PxOUT is 1 for pull-up
    //PxREN is 0 for no pull resistor
    //PxREN is 1 for pull resistor
    //PxIE is 0 for interrupt disable
    //PxIE is 1 for interrupt enable
    //PxIES is 0 for rising edge
    //PxIES is 1 for falling edge
    //PxIFG is 0 when no interrupt is pending
    //PxIFG is 1 when interrupt is pending
    //WARNING Writing to PxIES may set corresponding interrupt flag!!!

    //references a specific pin on a specific port
    struct portPin_s {
        const struct port_s * port;
        const uint8_t pin;
    };

    //stores the bit states for each register for a given GPIO configuration
    struct pin_mode_s {
        uint8_t dir;
        uint8_t out;
        uint8_t ren;
    };
    extern const struct pin_mode_s pin_mode_input;
    extern const struct pin_mode_s pin_mode_input_pullup;
    extern const struct pin_mode_s pin_mode_input_pulldown;
    extern const struct pin_mode_s pin_mode_output;

    //used so uninitialized ISRs return correctly
    void dummy(void);

    //stores the ISR callback for each pin on interrupt-enabled ports
    //not included in port_s struct so that ports that do not support interrupts do
not waste memory
    struct port_callbacks_s {
        void (*callbacks[8])(void);
    };
    extern struct port_callbacks_s port1_cb;
    extern struct port_callbacks_s port2_cb;
    extern struct port_callbacks_s port3_cb;
    extern struct port_callbacks_s port4_cb;

    //stores memory-mapped register addresses for a given port
    struct port_s {
        volatile uint8_t * dir_addr;
        volatile uint8_t * out_addr;
        volatile uint8_t * ren_addr;
        volatile uint8_t * in_addr;
        volatile uint8_t * ie_addr;
        volatile uint8_t * ies_addr;
        volatile uint8_t * ifg_addr;
        volatile uint8_t * sel0_addr;
        volatile uint8_t * sel1_addr;
        volatile uint8_t * selc_addr;
        struct port_callbacks_s * callbacks;
    };
    extern const struct port_s port1_v;
    extern const struct port_s port2_v;
    extern const struct port_s port3_v;
    extern const struct port_s port4_v;
    extern const struct port_s port5_v;
    extern const struct port_s port6_v;
    extern const struct port_s port7_v;
    extern const struct port_s port8_v;
    extern const struct port_s port9_v;
    extern const struct port_s port10_v;
    extern const struct port_s * port1;
    extern const struct port_s * port2;
    extern const struct port_s * port3;
    extern const struct port_s * port4;
    extern const struct port_s * port5;
    extern const struct port_s * port6;
    extern const struct port_s * port7;
    extern const struct port_s * port8;
    extern const struct port_s * port9;
    extern const struct port_s * port10;

    //bit and bitmask LUTs as a performance optimization, shifting is relatively
slow on the MSP430
    extern const uint8_t pinMasks[8];
    extern const uint8_t pinBits[8];

    //Convenience defines
    #define MODE_INPUT (&pin_mode_input)
    #define MODE_INPUT_PULLUP (&pin_mode_input_pullup)
    #define MODE_INPUT_PULLDOWN (&pin_mode_input_pulldown)
    #define MODE_OUTPUT (&pin_mode_output)

    #define P(X,Y) ((struct portPin_s){port##X, Y})
    //end convenience defines

#endif //end ARC_MSP_USE_GPIO

#ifdef ARC_MSP_USE_GPIO
    //configures an interrupt for a specific pin and optionally enables it
    void pinInterrupt(const struct portPin_s * portPin, uint8_t interruptEnable,
uint8_t edge, void (*isr_callback)(void));
    //convenience function to configure interrupts on a group of pins
    void group_pinInterrupt(const struct portPin_s (*portPins)[], const uint16_t
count, uint8_t interruptEnable, uint8_t edge, void (*isr_callback)(void));
    //convenience function to enable interrupt on a single pin
    void pinInterruptEnable(const struct portPin_s * portPin);
    //convenience function to enable interrupts on a group of pins
    void group_pinInterruptEnable(const struct portPin_s (*portPins)[], const
uint16_t count);
    //convenience function to disable interrupt on a single pin
    void pinInterruptDisable(const struct portPin_s * portPin);
    //convenience function to disable interrupts on a group of pins
    void group_pinInterruptDisable(const struct portPin_s (*portPins)[], const
uint16_t count);

    //configures a pin for GPIO using one of: MODE_INPUT, MODE_INPUT_PULLUP,
MODE_INPUT_PULLDOWN, MODE_OUTPUT
    void pinMode(const struct portPin_s * portPin, const struct pin_mode_s * mode);
    //convenience function to set a list of pins to the same GPIO mode
    void group_pinMode(const struct portPin_s (*portPins)[], const uint16_t count,
const struct pin_mode_s * mode);

    //selects a function for a GPIO pin
    void pinFunc(const struct portPin_s * portPin, const uint8_t func);
    //convenience function to set a list of pins to the same GPIO function
    void group_pinFunc(const struct portPin_s (*portPins)[], const uint16_t count,
const uint8_t func);

    //sets a given pin to either HIGH or LOW
    //WARNING: does NOT check if pin is set to output
    void digitalWrite(const struct portPin_s * portPin, uint8_t val);
    //convenience function to set a list of pins to either HIGH or LOW
    void group_digitalWrite(const struct portPin_s (*portPins)[], const uint16_t
count, uint8_t val);

    //reads the current state of a pin
    //WARNING: does NOT check if pin is set to input
    uint8_t digitalRead(const struct portPin_s * portPin);
    //reads the current state of a group of pins
    void group_digitalRead(const struct portPin_s (*portPins)[], const uint16_t
count, uint8_t (*result)[]);
#endif //end ARC_MSP_USE_GPIO

void arc_msp_setup(void);

#endif //end include guard
```

arc_msp_helper.c

```
/*
Andrew R. Courtemanche 2020/11
Updated 2020/12
*/

#define ARC_MSP_USE_ALL
#define ARC_MSP_TYPE_msp430fr6989
#include "arc_msp_helper.h"

#include <msp430.h>

//PxDIR is 0 for input
//PxDIR is 1 for output
//PxOUT is 0 for pull-down
//PxOUT is 1 for pull-up
//PxREN is 0 for no pull resistor
//PxREN is 1 for pull resistor
//PxIE is 0 for interrupt disable
//PxIE is 1 for interrupt enable
//PxIES is 0 for rising edge
//PxIES is 1 for falling edge
//PxIFG is 0 when no interrupt is pending
//PxIFG is 1 when interrupt is pending
//WARNING Writing to PxIES may set corresponding interrupt flag!!!

const struct pin_mode_s pin_mode_input =          {.dir = 0, .out = 0, .ren = 0};
const struct pin_mode_s pin_mode_input_pullup =   {.dir = 0, .out = 1, .ren = 1};
const struct pin_mode_s pin_mode_input_pulldown = {.dir = 0, .out = 0, .ren = 1};
const struct pin_mode_s pin_mode_output =         {.dir = 1, .out = 0, .ren = 0};

struct port_callbacks_s port1_cb = {{&dummy, &dummy, &dummy, &dummy, &dummy,
&dummy, &dummy, &dummy}};
struct port_callbacks_s port2_cb = {{&dummy, &dummy, &dummy, &dummy, &dummy,
&dummy, &dummy, &dummy}};
```

```c
struct port_callbacks_s port3_cb = {{&dummy, &dummy, &dummy, &dummy, &dummy,
&dummy, &dummy, &dummy}};
struct port_callbacks_s port4_cb = {{&dummy, &dummy, &dummy, &dummy, &dummy,
&dummy, &dummy, &dummy}};

const struct port_s port1_v  = {&P1DIR, &P1OUT, &P1REN, &P1IN, &P1IE, &P1IES,
&P1IFG, &P1SEL0, &P1SEL1, &P1SELC, &port1_cb}, * port1 =  &port1_v;
const struct port_s port2_v  = {&P2DIR, &P2OUT, &P2REN, &P2IN, &P2IE, &P2IES,
&P2IFG, &P2SEL0, &P2SEL1, &P2SELC, &port2_cb}, * port2 =  &port2_v;
const struct port_s port3_v  = {&P3DIR, &P3OUT, &P3REN, &P3IN, &P3IE, &P3IES,
&P3IFG, &P3SEL0, &P3SEL1, &P3SELC, &port3_cb}, * port3 =  &port3_v;
const struct port_s port4_v  = {&P4DIR, &P4OUT, &P4REN, &P4IN, &P4IE, &P4IES,
&P4IFG, &P4SEL0, &P4SEL1, &P4SELC, &port4_cb}, * port4 =  &port4_v;
const struct port_s port5_v  = {&P5DIR, &P5OUT, &P5REN, &P5IN, NULL, NULL, NULL,
&P5SEL0, &P5SEL1, &P5SELC, NULL},   * port5 =  &port5_v;
const struct port_s port6_v  = {&P6DIR, &P6OUT, &P6REN, &P6IN, NULL, NULL, NULL,
&P6SEL0, &P6SEL1, &P6SELC, NULL},   * port6 =  &port6_v;
const struct port_s port7_v  = {&P7DIR, &P7OUT, &P7REN, &P7IN, NULL, NULL, NULL,
&P7SEL0, &P7SEL1, &P7SELC, NULL},   * port7 =  &port7_v;
const struct port_s port8_v  = {&P8DIR, &P8OUT, &P8REN, &P8IN, NULL, NULL, NULL,
&P8SEL0, &P8SEL1, &P8SELC, NULL},   * port8 =  &port8_v;
const struct port_s port9_v  = {&P9DIR, &P9OUT, &P9REN, &P9IN, NULL, NULL, NULL,
&P9SEL0, &P9SEL1, &P9SELC, NULL},   * port9 =  &port9_v;
const struct port_s port10_v = {&P10DIR, &P10OUT, &P10REN, &P10IN, NULL, NULL,
NULL, &P10SEL0, &P10SEL1, &P10SELC, NULL},  * port10 = &port10_v;

//bit and bitmask LUTs as a performance optimization, shifting is relatively slow
on the MSP430
const uint8_t pinMasks[8] = {(uint8_t)~(0x1<<0), (uint8_t)~(0x1<<1),
(uint8_t)~(0x1<<2), (uint8_t)~(0x1<<3), (uint8_t)~(0x1<<4), (uint8_t)~(0x1<<5),
(uint8_t)~(0x1<<6), (uint8_t)~(0x1<<7)};
const uint8_t pinBits[8] =  { (0x1<<0),  (0x1<<1),  (0x1<<2),  (0x1<<3),  (0x1<<4),
(0x1<<5),  (0x1<<6),  (0x1<<7)};

//Looks up and calls the specific callback for a given pin.
//This is really too much work for an ISR, but it's mighty convenient
static void ISR_HANDLER(const struct port_s * port ) {
    uint8_t ifg = *(port->ifg_addr); //get interrupt flags for this port
    uint8_t i=0;
    for (; i<8; i++) { //check all 8 flags for this port
        if (ifg & 0x1) { //check if flag is set
            (port->callbacks->callbacks[i])(); //call callback
        }
        ifg = ifg >> 1; //get next flag
    }
    *(port->ifg_addr) = 0; //clear interrupt flag
}

//Interrupt Service Routines for each port
__attribute__ ((interrupt(PORT1_VECTOR)))
static void Port_1(void) {
    ISR_HANDLER(port1);
}
__attribute__ ((interrupt(PORT2_VECTOR)))
static void Port_2(void) {
    ISR_HANDLER(port2);
}
__attribute__ ((interrupt(PORT3_VECTOR)))
static void Port_3(void) {
    ISR_HANDLER(port3);
}
__attribute__ ((interrupt(PORT4_VECTOR)))
static void Port_4(void) {
    ISR_HANDLER(port4);
}

void dummy(void){
    return;
}

//configures an interrupt for a specific pin and optionally enables it
void pinInterrupt(const struct portPin_s * portPin, uint8_t interruptEnable,
uint8_t edge, void (*isr_callback)(void)) {
    if (portPin->port->callbacks == NULL) return; //early return if port is not
interrupt capable
    //_disable_interrupt();
    *(portPin->port->ie_addr) = (*(portPin->port->ie_addr) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * interruptEnable); //optionally enable interrupt
    *(portPin->port->ies_addr) = (*(portPin->port->ies_addr) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * edge); //configure trigger edge
    *(portPin->port->ifg_addr) = (*(portPin->port->ifg_addr) & pinMasks[portPin-
>pin]); //clear ifg, just in case
    portPin->port->callbacks->callbacks[portPin->pin] = isr_callback; //store
callback
    //_enable_interrupt();
}
//convenience function to configure interrupts on a group of pins
void group_pinInterrupt(const struct portPin_s (*portPins)[], const uint16_t count,
uint8_t interruptEnable, uint8_t edge, void (*isr_callback)(void)) {
    uint8_t i=0;
    for (; i<count; i++) {
        pinInterrupt(&(*portPins)[i], interruptEnable, edge, isr_callback);
    }
}
//convenience function to enable interrupt on a single pin
void pinInterruptEnable(const struct portPin_s * portPin) {
    if (portPin->port->callbacks == NULL) return; //early return if port is not
interrupt capable
    //_disable_interrupt();
    *(portPin->port->ie_addr) = (*(portPin->port->ie_addr) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin]); //enable interrupt
    *(portPin->port->ifg_addr) = (*(portPin->port->ifg_addr) & pinMasks[portPin-
>pin]); //clear ifg
    //_enable_interrupt();
}
//convenience function to enable interrupts on a group of pins
void group_pinInterruptEnable(const struct portPin_s (*portPins)[], const uint16_t
count) {
    uint8_t i=0;
    for (; i<count; i++) {
        pinInterruptEnable(&(*portPins)[i]);
    }
}
//convenience function to disable interrupt on a single pin
void pinInterruptDisable(const struct portPin_s * portPin) {
    if (portPin->port->callbacks == NULL) return; //early return if port is not
interrupt capable
    //_disable_interrupt();
    *(portPin->port->ie_addr) = (*(portPin->port->ie_addr) & pinMasks[portPin-
>pin]); //disable interrupt
    *(portPin->port->ifg_addr) = (*(portPin->port->ifg_addr) & pinMasks[portPin-
>pin]); //clear ifg
    //_enable_interrupt();
}
//convenience function to disable interrupts on a group of pins
void group_pinInterruptDisable(const struct portPin_s (*portPins)[], const uint16_t
count) {
    uint8_t i=0;
    for (; i<count; i++) {
        pinInterruptDisable(&(*portPins)[i]);
    }
}

//configures a pin for GPIO using one of: MODE_INPUT, MODE_INPUT_PULLUP,
MODE_INPUT_PULLDOWN, MODE_OUTPUT
void pinMode(const struct portPin_s * portPin, const struct pin_mode_s * mode) {
    *(portPin->port->dir_addr) = ((*(portPin->port->dir_addr)) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * mode->dir);
    *(portPin->port->out_addr) = ((*(portPin->port->out_addr)) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * mode->out);
    *(portPin->port->ren_addr) = ((*(portPin->port->ren_addr)) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * mode->ren);
}
//convenience function to set a list of pins to the same GPIO mode
void group_pinMode(const struct portPin_s (*portPins)[], const uint16_t count,
const struct pin_mode_s * mode) {
    uint8_t i=0;
    for (; i<count; i++) {
        pinMode(&(*portPins)[i], mode);
    }
}

//selects a function for a GPIO pin
void pinFunc(const struct portPin_s * portPin, const uint8_t func) {
    *(portPin->port->sel0_addr) = ((*(portPin->port->sel0_addr)) &
pinMasks[portPin->pin]) | (pinBits[portPin->pin] * (func & 0b01));
    *(portPin->port->sel1_addr) = ((*(portPin->port->sel1_addr)) &
pinMasks[portPin->pin]) | (pinBits[portPin->pin] * ((func & 0b10)>>1));
    //*(portPin->port->selc_addr) = ((*(portPin->port->selc_addr)) &
pinMasks[portPin->pin]) | (pinBits[portPin->pin] * (func & 0b11)); //dunno
}
//convenience function to set a list of pins to the same GPIO function
void group_pinFunc(const struct portPin_s (*portPins)[], const uint16_t count,
const uint8_t func) {
    uint8_t i=0;
    for (; i<count; i++) {
        pinFunc(&(*portPins)[i], func);
    }
}

//sets a given pin to either HIGH or LOW
//WARNING: does NOT check if pin is set to output
void digitalWrite(const struct portPin_s * portPin, uint8_t val) {
    *(portPin->port->out_addr) = ((*(portPin->port->out_addr)) & pinMasks[portPin-
>pin]) | (pinBits[portPin->pin] * val);
}
//convenience function to set a list of pins to either HIGH or LOW
void group_digitalWrite(const struct portPin_s (*portPins)[], const uint16_t count,
uint8_t val) {
    uint8_t i=0;
    for (; i<count; i++) {
        digitalWrite(&(*portPins)[i], val);
    }
}

//reads the current state of a pin
//WARNING: does NOT check if pin is set to input
uint8_t digitalRead(const struct portPin_s * portPin) {
    return ((*(portPin->port->in_addr)) & pinBits[portPin->pin]) >> portPin->pin;
}
//reads the current state of a group of pins
void group_digitalRead(const struct portPin_s (*portPins)[], const uint16_t count,
uint8_t (*result)[]) {
    uint8_t i=0;
    for (; i<count; i++) {
        (*result)[i] = digitalRead(&(*portPins)[i]);
    }
}

void arc_msp_setup(void) {
    PM5CTL0 &= ~LOCKLPM5;
}
```