ME241 – Introduction to Computer Structures & Real-Time Systems

Lab Project 1: The Scaffold of our RTOS

Prepared by:

Nathan Tang (20890522)

Eddy Zhou (20873917)

Submitted on:

21 September 2022

Part 1: Creating Two Stacks

In the following files `_threadsCore.h` and `_threadsCore.c`, 3 functions were declared and implemented. `uint32_t* getMSPInitialLocation(void)` is used to identify the initial memory location of MSP via the vector table at address 0x0. `uint32_t* getNewThreadStack(uint32_t offset)` returns a new PSP that is decremented by an 'offset' number of specified bytes passed as a parameter. `Void setThreadingWithPSP(uint32_t* threadStack)` sets the PSP to the pointer passed and switched to PSP via the control bit. To test the following functions, the debugger was used.

Starting a debugging session, and placing a breakpoint at the line where the infinite loop is run in the main function allows for the examination of memory content after the contents in the main function have executed. Running to the following breakpoint yields the following register values.

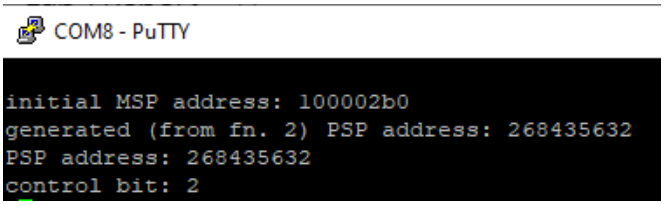

Figure 1. Console after executing program for Part 1

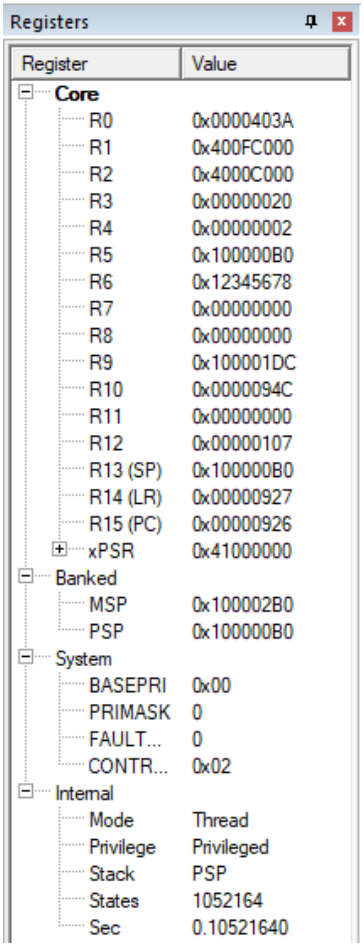

Figure 2. Register Values after Running Debugger Session for Part 1

Figure 1 shows the console messages in PuTTY when using the previously mentioned functions to retrieve the initial MSP location, create a PSP to a new stack that is 512 bytes offset from MSP, and switch to using PSP. Figure 2 shows the register content of the following process. As expected, the difference between MSP and PSP is 512 bytes, CONTROL is set to 0x02, and Mode is set to “Thread”.

Part 2: The PendSV Interrupt

The PendSV Interrupt is an interrupt service routine which only runs once any running interrupt has finished. In this lab, a simple assembly-based PendSV ISR was made. This ISR is first triggered by a C function, where a debugger is used to read the state of the registers at a breakpoint in the routine.

Premise

The general premise of the code below is to first create an interrupt handler, and then call the handler on purpose via triggering a software interrupt. The handler itself is pretty straightforward. It instructs the chip to move a special value into the link register which, when branched into, will make the chip return back to Thread mode and revert back to using the PSP. In order to test if this interrupt handler works, a software-based interrupt is triggered by two functions `void kernelInit(void)` and `void osSched(void)` which, for now, are not setup to do exactly what they are supposed to do. Instead, these two functions contain only the minimum amount of code needed to set the priority of the PendSV interrupt and also trigger the interrupt.

Analysis of svc_call.s

In this file, the PendSV handler is created. In brief, the ARM Cortex-M is instructed to return to Thread mode and revert back to using the PSP whenever the PendSV_Handler is called..

```
AREA  handle_pend,CODE,READONLY
```

This line instructs the executable to construct a new section of code called ‘handle_pend’. This line also tells the executable that ‘handle_pend’ is read-only and hence cannot be overwritten.

```
GLOBAL  PendSV_Handler
```

Because the PendSV_handler will be a function referenced to handle interrupts from other pieces of code written elsewhere, this function should be declared globally. This line simply defines that there is a will be a global function with name PendSV_Handler

```
PRESERVE8
```

This line specifies that the file preserves 8-byte alignment. This means that all objects are stored in memory addresses that are a multiple of 8 [1].

```
PendSV_Handler
```

This line actually begins the function definition of the PendSV_Handler.

```
MOV LR,#0xFFFFFFFD
```

Loads 0xFFFFFFFD into the link register. This special value tells the chip to return from an ISR, return from handler mode to Thread mode, and to revert back to the PSP from the MSP.

```
BX LR
```

This line instructs the chip to branch to LR (which contains the special code) while maintaining important state information in the processor.

```
END
```

Indicates the end of the assembly file.

Analysis of Performing the Interrupt

In order to trigger the interrupt handler, an interrupt needs to be triggered. One method is to trigger a software interrupt which consists of writing a function to tell the chip that an interrupt approaches, writing to a memory location meant for interrupts, and allowing the interrupt to complete, returning from the function if needed.

_kernelCore.h

```
#define SHPR3 *(uint32_t*)0xE000ED20
```

Defines the location of the PendSV priority register. This memory location allows us to change the priority of the PendSV Interrupt according to the bits we assign to it.

```
#define ICSR *(uint32_t*)0x E000ED04
```

Defines the location of the Interrupt control and State Register. This memory location allows us to set the PendSV exception state to pending.

_kernelCore.c

Inside void kernelInit(void)

```
SHPR3 |= 0xFF << 16
```

We are assigning bits 23-16 to the highest possible number which in turn represents the lowest priority.

Inside void osSched(void)

```
ICSR |= 1<<28;
```

We are assigning bit 28 to 1 which sets the PendSV exception state to pending.

```
__asm("isb");
```

Runs the ISB instruction using assembly. This line instructs the chip to flush the instruction pipeline [2].

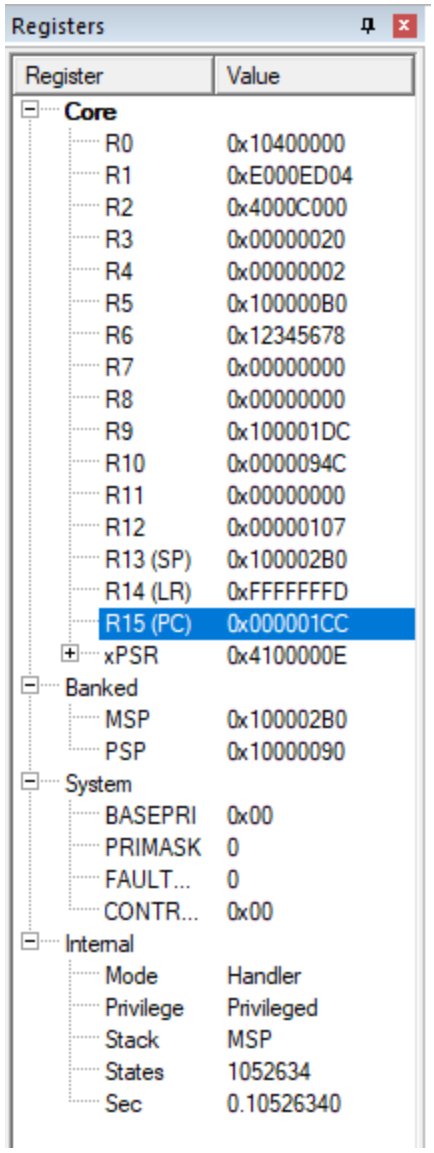

Figure 3. Register Values after Running Debugger Session Inside the PendSV Interrupt

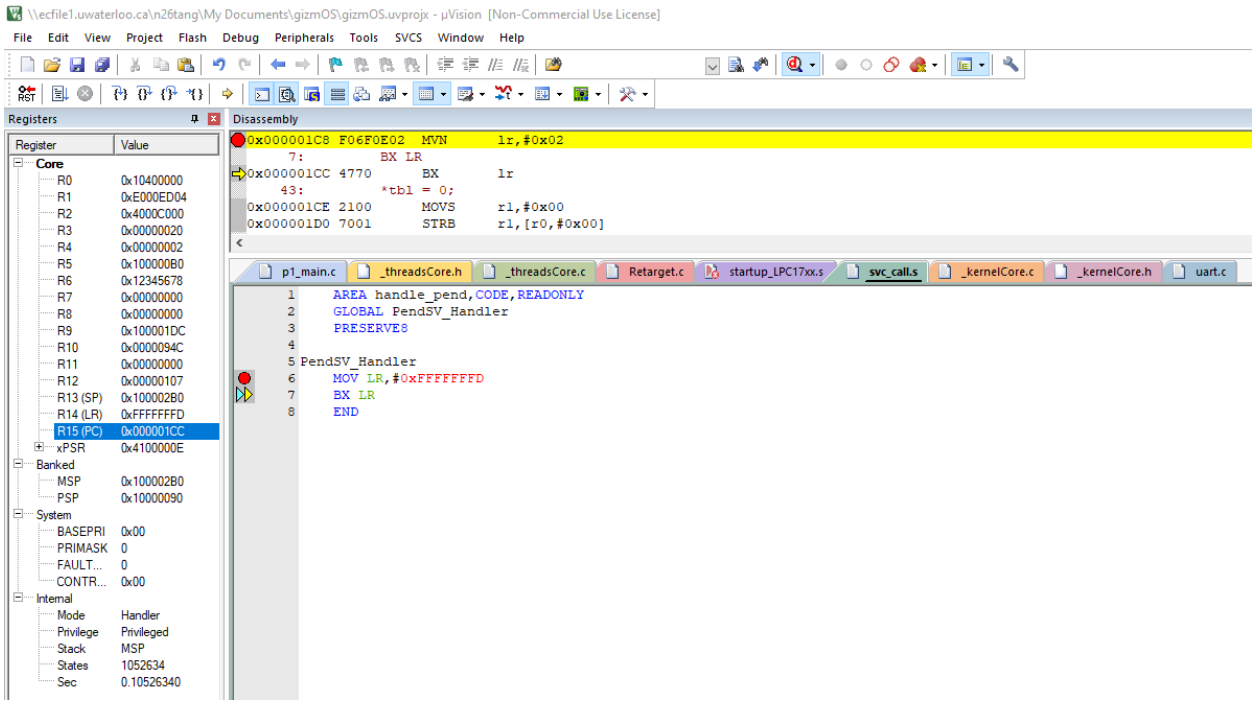

Figure 4. Entire Debugger Session Inside the PendSV Interrupt

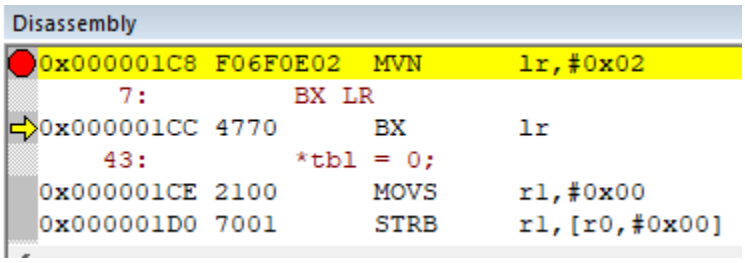

Figure 5. Disassembly after Running Debugger Session Inside the PendSV Interrupt

References

[1] S. H. Ahn, “Data Alignment,” *Data alignment*, 2012. [Online]. Available: http://www.songho.ca/misc/alignment/dataalign.html. [Accessed: 21-Sep-2022].

[2] “ISB,” *Documentation – arm developer*. [Online]. Available: https://developer.arm.com/documentation/dui0473/m/arm-and-thumb-instructions/isb. [Accessed: 21-Sep-2022].